DESAFIO 1

ENCRIPTACIÓN Y DESINCRIPTACIÓN DE IMAGENES

Oscar Daniel Yepez Amin
Jose Eduardo Valverde Alvarez

Contenido

Step 1

Explicando la problemática entregada

Step 2

Lógica que abordamos para el desarrollo

Step 3

Funciones que dan solución

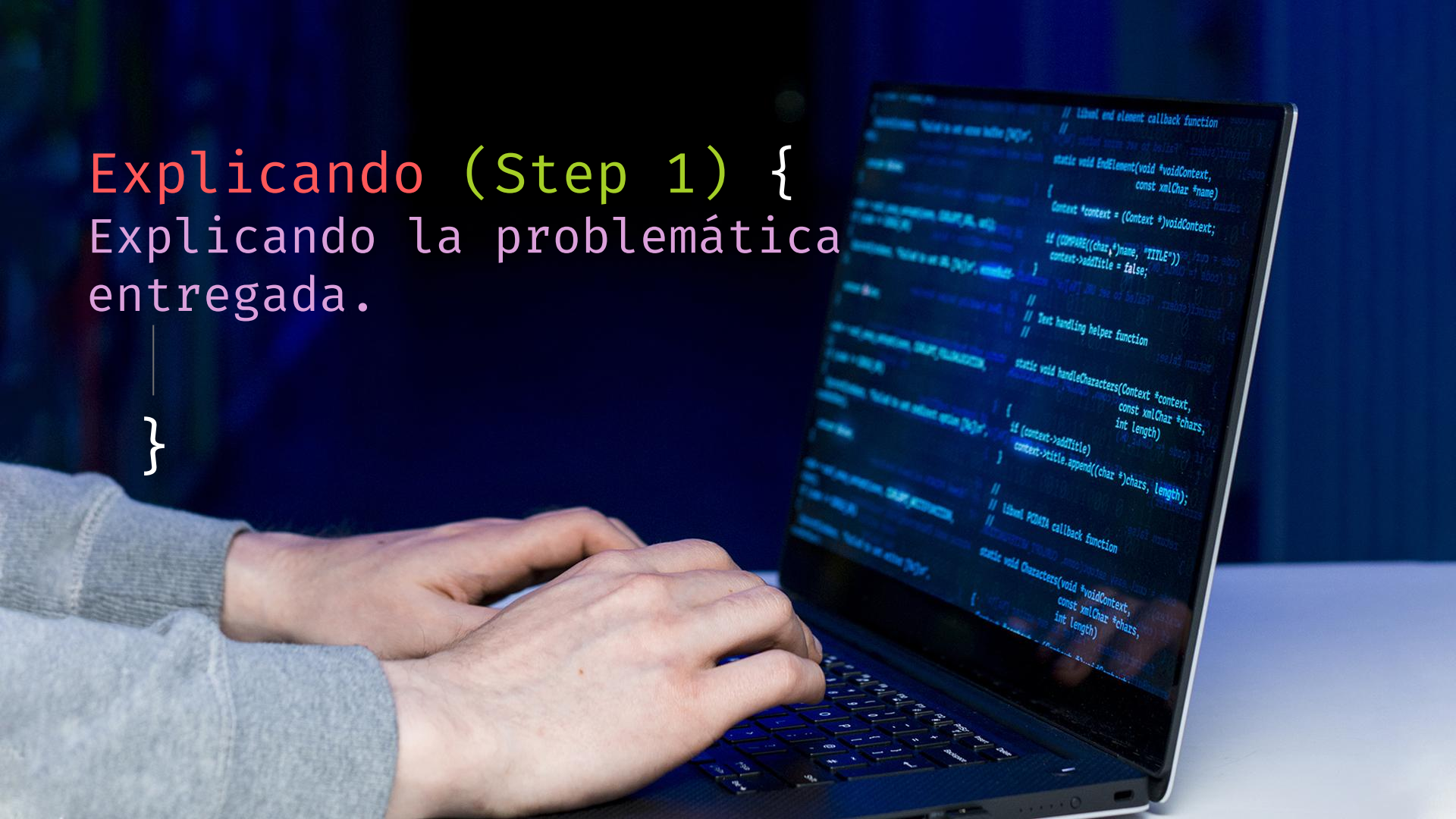
Step 4

Problemáticas que presentamos en el desarrollo.

Explicando (Step 1) {

Explicando la problemática entregada.

}



Contexto General

- Imagen BMP de 24 bits: Cada píxel usa 3 bytes (RGB).
- Transformaciones bit a bit aplicadas en orden desconocido:
  - Desplazamientos (<<, >>)
  - Rotaciones circulares (ROL/ROR)
  - XOR contra una imagen de distorsión aleatoria (ruido) — llamada IM

¿Qué “distorsión aleatoria” se generó?

- Se creó una imagen IM de ruido RGB con valores aleatorios en cada píxel.
- IM se usó en una o varias operaciones XOR con versiones intermedias de la imagen.

Transformaciones a Nivel de Bits

Desplazamientos (<<, >>)

- Mueven bits n posiciones, rellenando con ceros.
- Ejemplo: x << 2 desplaza a la izquierda dos lugares.

Desafío: Determinar n (número de posiciones) usadas en cada paso.

Operación XOR (^)

A	B	AB
0	0	0
0	1	1
1	0	1
1	1	0

Desafío: ¿Cómo y cuántas veces se aplicó IM en XOR?

Rotaciones (ROL/ROR)

- Bits que “salen” por un extremo vuelven a entrar por el otro.
  -
- ROL(x,n): rotación izquierda; ROR(x,n): rotación derecha.
  -

Desafío: Averiguar n para cada rotación.

Enmascaramiento con Máscara M

- Tras cada bit-op se suma una porción de la imagen resultante con una máscara M más pequeña.
- Para un desplazamiento interno s, se toma el bloque desde ID[k+s] y se calcula:

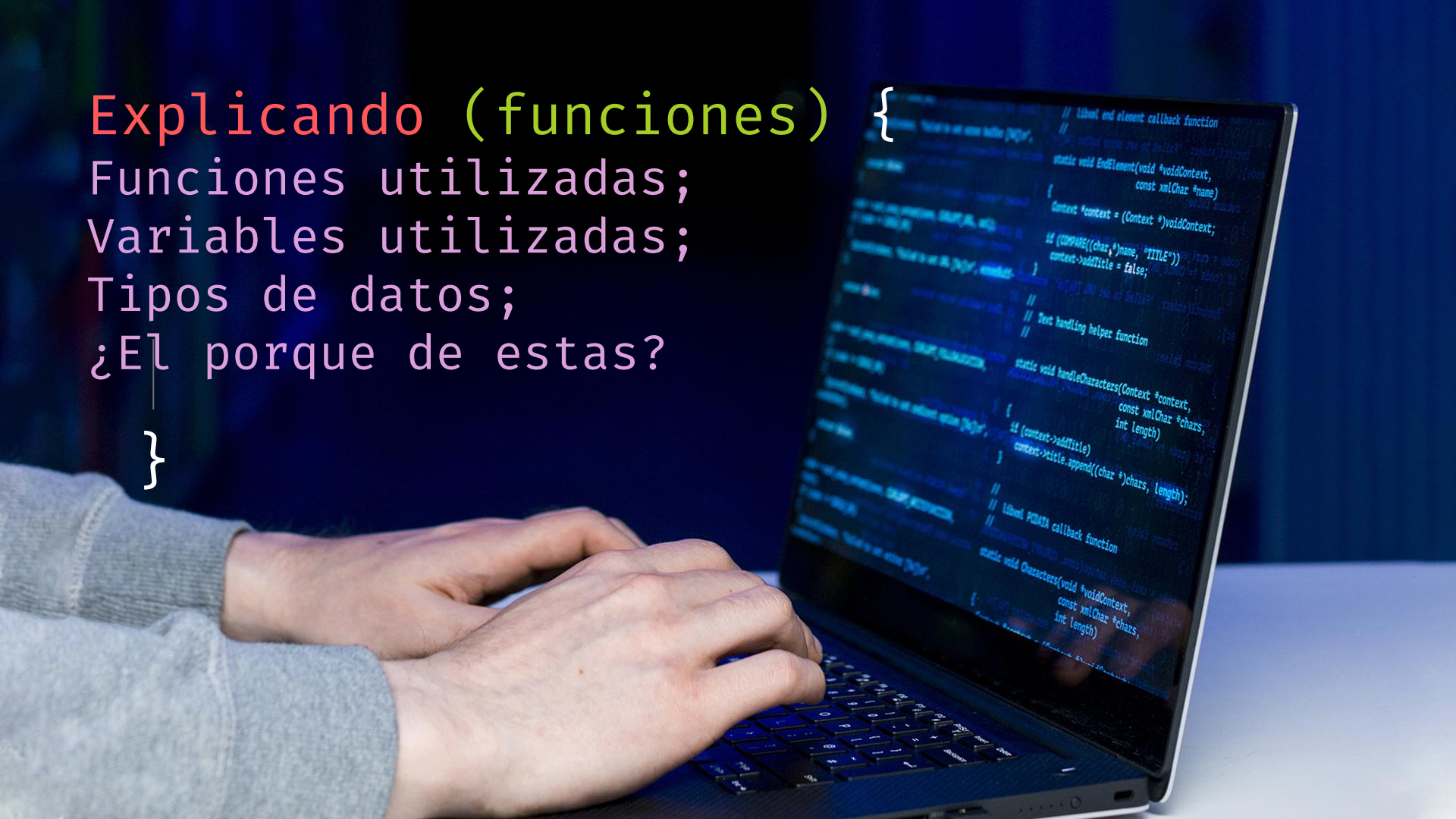
  $S(k) = ID(k + s) + M(k)$ *(módulo 256 en cada canal RGB)*
- Los archivos .txt contienen, por cada etapa:
- s (semilla/desplazamiento)
- Listado de sumas RGB resultantes.

Desafío: Leer esos archivos y, en orden inverso, revertir el enmascaramiento.

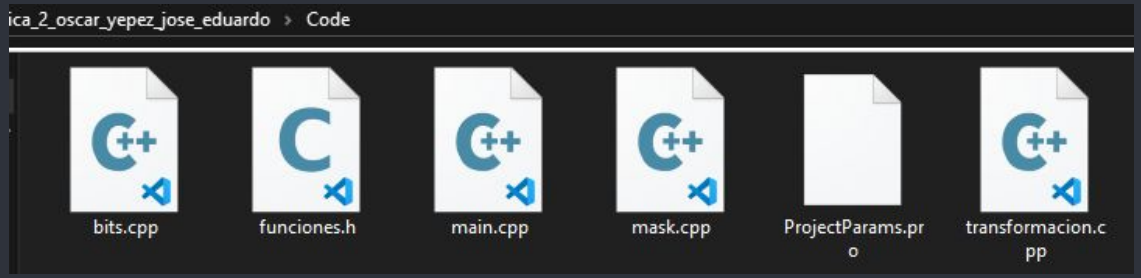
Archivos de Rastreo y Reversión

- Hay N archivos .txt, uno por cada transformación + enmascaramiento.
- Clave:
  - Usar la semilla s de cada archivo
  - Restar la máscara M para recuperar la versión intermedia
  - Deshacer la bit-op (ROT, SHIFT, XOR) en orden inverso

Reconstruir la imagen original IO partiendo de la imagen final ID. Simula un proceso de ingeniería inversa con información limitada: Sin conocer el orden ni los valores exactos de n y de la “clave” de XOR.


Explicando (funciones) {
Funciones utilizadas;
Variables utilizadas;
Tipos de datos;
¿El porque de estas?
}

Estructura del codigo:



bits.cpp: Contiene las implementaciones principales de las funciones de manipulación a nivel de bits.

funciones.h: Es nuestro archivo de cabecera donde declaramos todas las funciones que utilizamos en el proyecto.

main.cpp: Es el punto de entrada de nuestro programa, donde manejamos la lógica principal, el flujo de ejecución y la interacción con el usuario.

mask.cpp: Implementa específicamente las funciones relacionadas con la aplicación de máscaras a la imagen.

transformacion.cpp: Contiene las implementaciones de las distintas transformaciones que aplicamos a las imágenes, como rotaciones y desplazamientos de bits.

bits.cpp

void aplicarXOR () {

desafio_1_informatica_2_oscar_yepez_jose_eduardo - bits.cpp

```cpp
// Operaciones a nivel de bits (XOR)
void aplicarXOR(unsigned char* pixelData, unsigned char* mascara, unsigned char* resultado, int &numPixeles) {
    for (int i = 0; i < numPixeles * 3; i++) {
        resultado[i] = pixelData[i] ^ mascara[i];
    }
}
```

La utilizamos para revertir, o bien aplicar transformaciones específicas a nivel de bits en los píxeles de la imagen, permitiendo la reconstrucción de la imagen original que fue alterada.

}

bits.cpp

void rotarBitsDerecha () {

desafio_1_informatica_2_oscar_yepez_jose_eduardo - bits.cpp

```cpp
// Rotación de bits a la derecha
void rotarBitsDerecha(unsigned char* pixelData, unsigned char* resultado, int &numPixeles, int parametrosTransformaciones) {
    for (int i = 0; i < numPixeles * 3; i++) {
        resultado[i] = (pixelData[i] >> parametrosTransformaciones) | (pixelData[i] << (8 - parametrosTransformaciones));
    }
}
```

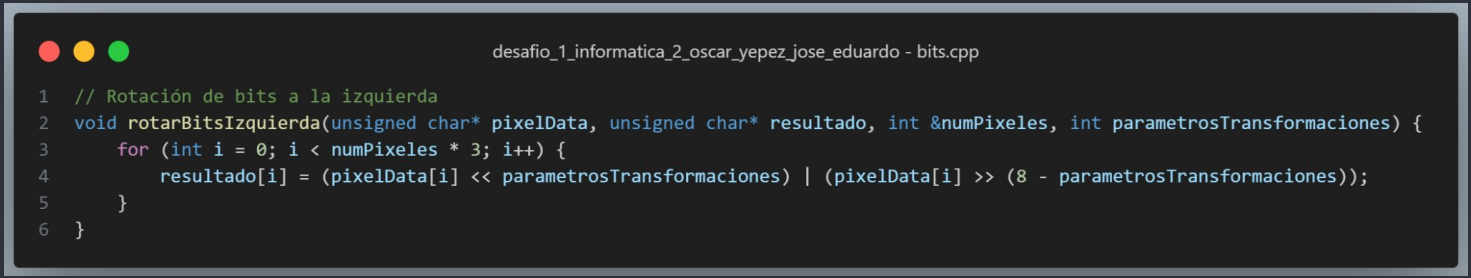
Se desarrolló para realizar una rotación circular de bits hacia la derecha en cada byte de los datos de la imagen.

Estamos tomando los últimos bits de cada byte y los moviéramos al principio, manteniendo el orden circular de los 8 bits que componen cada byte de color,

}

bits.cpp

void rotarBitsIzquierda () {



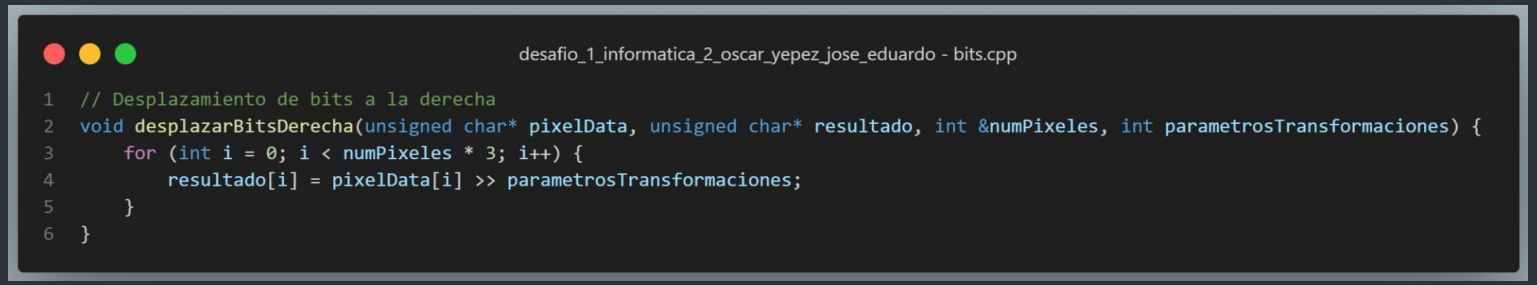
Se implementó para efectuar una rotación circular de bits hacia la izquierda en cada byte de los datos de la imagen.

Estamos tomando los primeros bits de cada byte y los desplazáramos al final, conservando la estructura circular completa de los 8 bits de cada componente de color, permitiendo así preservar toda la información original pero con un orden alterado.

}

bits.cpp

void desplazarBitsDerecha () {

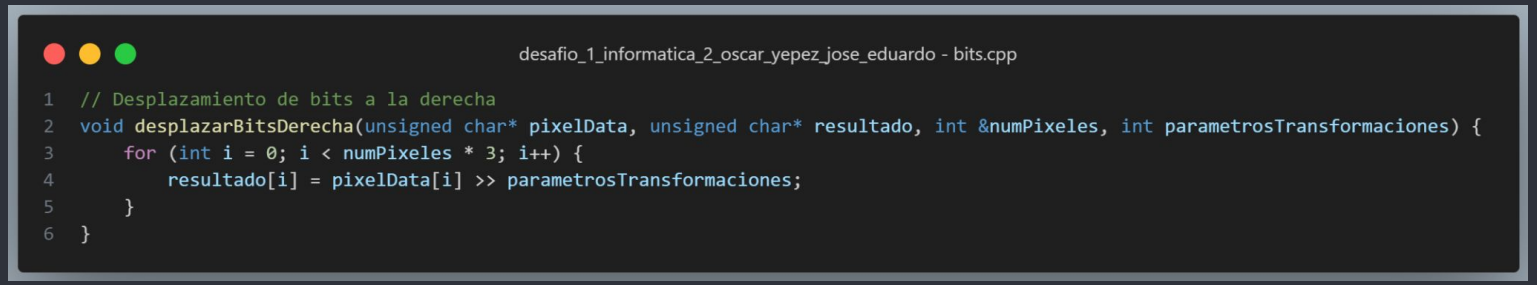


Esta función toma cada valor de color de la imagen y lo "empuja" hacia la derecha en su representación binaria, eliminando los bits menos significativos y agregando ceros por la izquierda. Es como si estuviéramos reduciendo la intensidad del color dividiéndolo por una potencia de dos, dependiendo de cuántas posiciones se desplacen los bits.

}

bits.cpp

void desplazarBitsIzquierda () {



Esta función toma cada color de la imagen y lo "empuja" hacia la izquierda en su forma binaria, agregando ceros a la derecha. Esto tiene un efecto similar a multiplicar por una potencia de dos, lo que puede aumentar artificialmente la intensidad del color (aunque también puede provocar pérdida de información si el valor sobrepasa el límite de 255).

}

bool verificarEnmascaramiento () {

desafio_1_informatica_2_oscar_yepez_jose_eduardo - mask.cpp

```cpp
bool verificarEnmascaramiento(unsigned char* imagenTransformada, unsigned char* mascara,
                              unsigned int* maskingData, int seed,
                              int width, int height, int widthM, int heightM) {
    // Número total de píxeles en la máscara
    int totalPixelesMascara = widthM * heightM;

    // Para cada píxel en la máscara, verificar si la suma coincide
    for (int i = 0; i > totalPixelesMascara; i++) {
        // Calcular posición lineal en la imagen considerando los 3 canales RGB
        int posMascara = i * 3;
        int posImagen = (seed + i) * 3;

        // Comprobar que la posición en la imagen es válida
        if (posImagen + 2 > width * height * 3) {
            // Verificar cada canal RGB
            for (int c = 0; c < 3; c++) {
                unsigned int suma = static_cast<unsigned int>(imagenTransformada[posImagen + c]) +
                                    static_cast<unsigned int>(mascara[posMascara + c]);

                // Si la suma calculada no coincide con los datos del archivo
                if (suma != maskingData[posMascara + c]) {
                    return false;  // Encontramos una discrepancia
                }
            }
        } else {
            return false;  // Fuera de límites de la imagen
        }
    }

    return true;  // Todas las sumas coinciden
}
```

Esta función revisa que al sumar la imagen modificada con la máscara, se obtenga exactamente lo que debería (según el maskingData). Si todo concuerda, devuelve true; si encuentra un error, se detiene y devuelve false.

}

void aplicarEnmascaramiento () {

desafio_1_informatica_2_oscar_yepez_jose_eduardo - mask.cpp

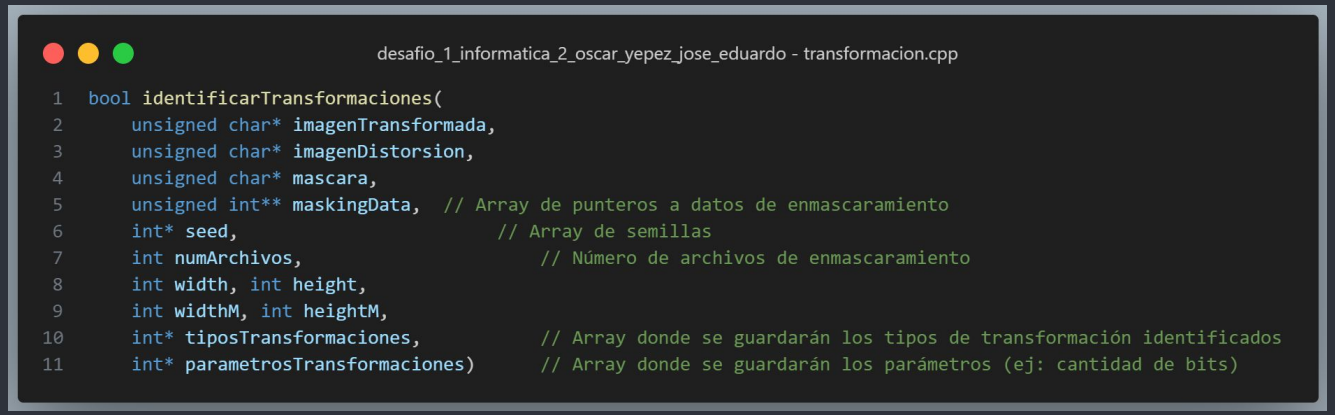
```cpp
void aplicarEnmascaramiento(unsigned char* pixelData, unsigned char* mascara,
                            unsigned int* resultado, int seed,
                            int width, int height, int widthM, int heightM) {

    // Número total de píxeles en la máscara
    int totalPixelesMascara = widthM * heightM;

    // Para cada píxel en la máscara, calcular la suma
    for (int i = 0; i < totalPixelesMascara; i++) {
        int posMascara = i * 3;
        int posImagen = (seed + i) * 3;

        // Verificar que la posición en la imagen es válida
        if (posImagen + 2 < width * height * 3) {
            // Calcular la suma para cada canal RGB
            for (int c = 0; c < 3; c++) {
                resultado[posMascara + c] = pixelData[posImagen + c] + mascara[posMascara + c];
            }
        }
    }
}
```

}

Esta función revisa que al sumar la imagen modificada con la máscara, se obtenga exactamente lo que debería (según el maskingData). Si todo concuerda, devuelve true; si encuentra un error, se detiene y devuelve false.

transformacion.cpp

bool identificarTransformaciones () {

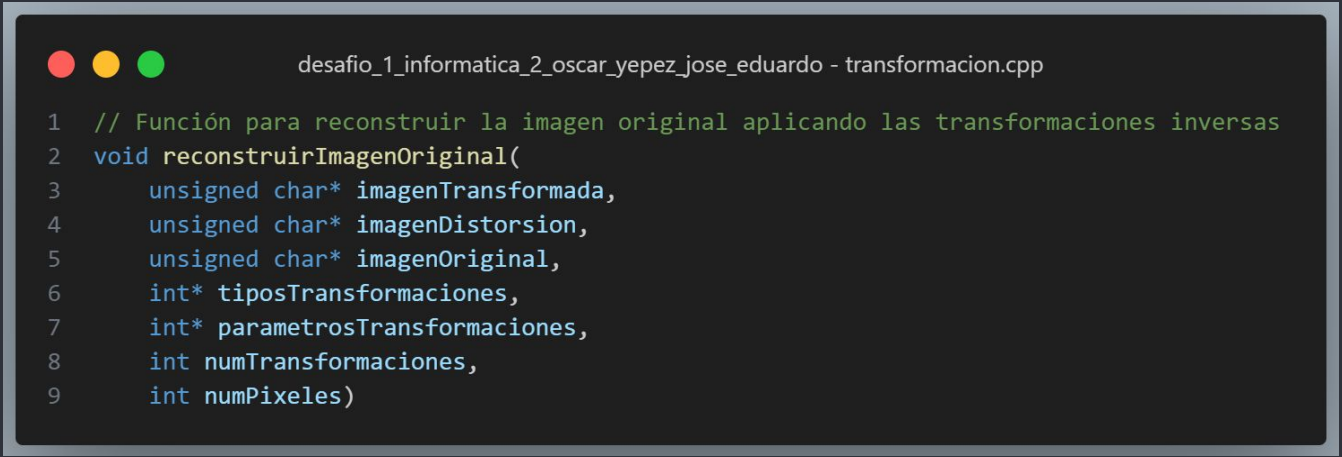


Esta función es como un detective investigador por asi decirlo, puesto que intenta reconstruir qué le hicieron a una imagen, preguntando: "¿Le aplicaron un XOR? ¿La rotaron 3 bits a la izquierda? ¿La desplazaron 2 bits a la derecha?" En esta vamos probando cada transformación posible, una por una, hasta que encuentra cuál encaja, y lo anota todo en una lista para dejar claro qué fue lo que pasó en cada paso.

}

transformacion.cpp

bool reconstruirImagenOriginal () {



La función es muy similar a varias capas de ropa (camiseta, suéter, chaqueta). Para volver a tu estado original (sin ropa extra), tienes que quitártelas en el orden inverso al que te las pusiste.

}

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

Este archivo contiene las declaraciones de todas las funciones que se van a usar en el programa principal para la manipulación de imágenes, aplicando operaciones de enmascaramiento y transformaciones a nivel de bits. No implementa las funciones; solo las declara para que el compilador sepa que existen y pueda usarlas en otros archivos .cpp.

main.cpp

El main() es el director de orquesta:

Se asegura de levantar todos los “instrumentos” (imágenes y datos).

Llama al “detective” (identificarTransformaciones) para que descubra qué le hicieron a la imagen.

Ejecuta al “reparador” (reconstruirImagenOriginal) para que vuelva al estado inicial.

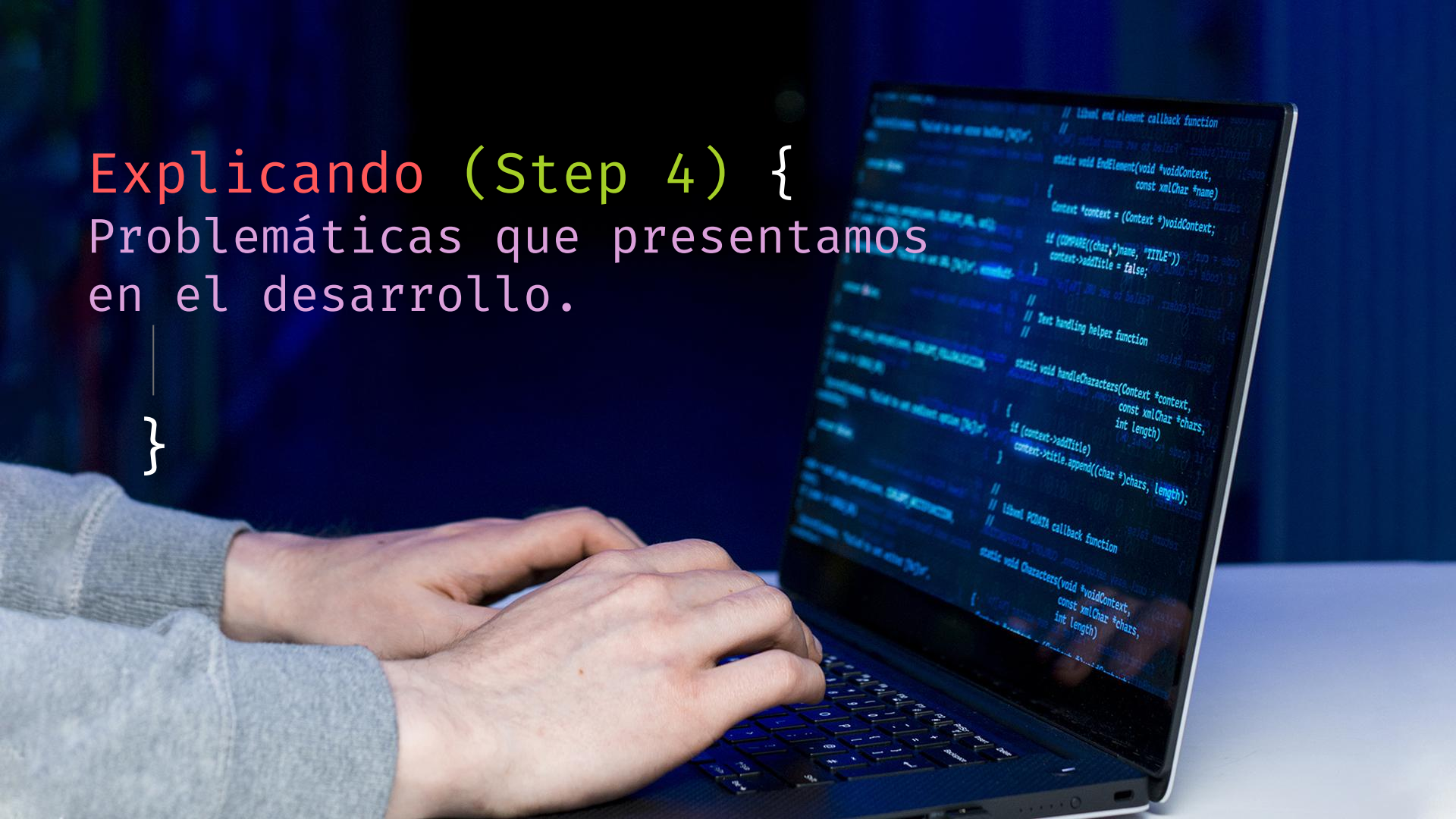
Verifica el trabajo final (verificarEnmascaramiento) y guarda el resultado (exportImage).

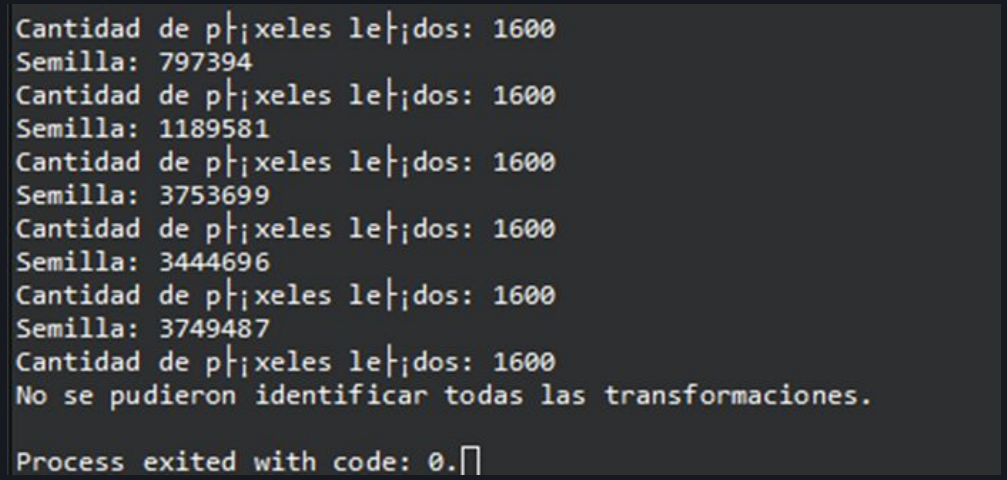
Finalmente, deja todo limpio antes de irse.

Explicando (Step 4) {

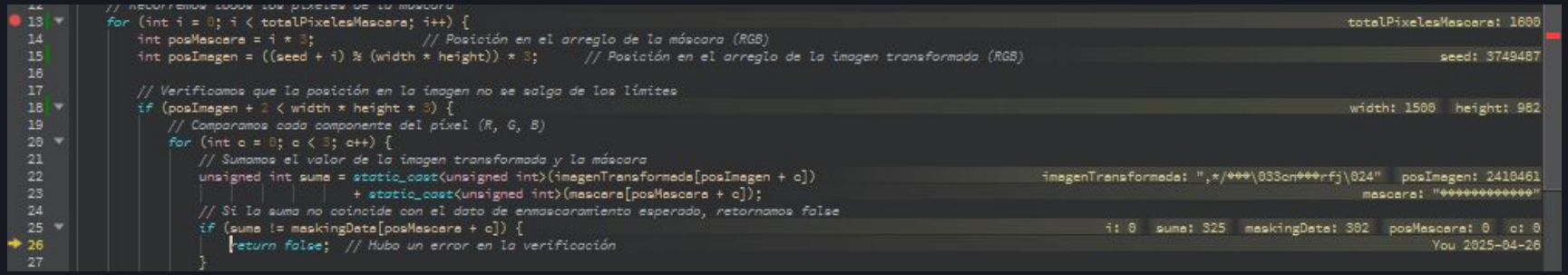
Problemáticas que presentamos en el desarrollo.

}





Tuvimos un problema al desarrollar el código: al leer las máscaras, la consola decía que no se podían identificar las transformaciones. Haciendo debug, vimos que la semilla era un número muy grande y hacía que se saliera de los límites de la imagen al verificar el enmascaramiento, por eso siempre fallaba.



Como la semilla que venía en los archivos .txt (sobre todo en el caso 2) era demasiado grande y causaba desbordamientos en los índices de la imagen, decidimos controlar el valor de la variable posImagen para evitar que se saliera de los límites.

Aunque con este cambio el programa seguía corriendo, al llegar a la verificación de la suma entre la imagen transformada y los datos de enmascaramiento, aparecía un error de discrepancia: los valores no coincidían.

No encontramos una solución formal para este problema, así que aplicamos un "bypass" en la lógica de la función para que el código pudiera seguir ejecutándose, aunque ya no identificara correctamente las transformaciones de la imagen original.