Steganography Image Encoder and Decoder

```cpp
#include <iostream>
#include <fstream>
#include <bitset>
using namespace std;

void setLSB(char &byte, int bit) {
    byte = (byte & 0xFE) | (bit & 1);
}

int getLSB(char byte) {
    return byte & 1;
}

void encodeMessageInImage(const string &imageFile, const string &message,
const string &outputFile) {
    ifstream image(imageFile, ios::binary);
    ofstream output(outputFile, ios::binary);

    if (!image.is_open() || !output.is_open()) {
        cerr << "Error opening file!" << endl;
        return;
    }

    char header[54];
    image.read(header, 54);
    output.write(header, 54);

    string binaryMessage;
    for (char c : message) {
        binaryMessage += bitset<8>(c).to_string();
    }

    binaryMessage += "00000000";

    char pixel;
    size_t messageIndex = 0;

    while (image.get(pixel)) {
        if (messageIndex < binaryMessage.size()) {
            setLSB(pixel, binaryMessage[messageIndex] - '0');
            messageIndex++;
        }
        output.put(pixel);
    }

    image.close();
    output.close();
    cout << "Message encoded in " << outputFile << endl;
}

string decodeMessageFromImage(const string &imageFile) {
    ifstream image(imageFile, ios::binary);

    if (!image.is_open()) {
        cerr << "Error opening file!" << endl;
        return "";
    }

    image.seekg(54);

    string binaryMessage;
    char pixel;
    string message;

    while (image.get(pixel)) {
        binaryMessage += to_string(getLSB(pixel));

        if (binaryMessage.size() == 8) {
            char decodedChar = bitset<8>(binaryMessage).to_ulong();
            if (decodedChar == '\0') {
                break;
            }
            message += decodedChar;
            binaryMessage.clear();
        }
    }
    image.close();
    return message;
}

int main() {
    string imageFile = "input.bmp";
    string outputFile = "output.bmp";
    string message;
    int choice;
    do{
        printf("1. Encode\n2. Decode\n3. Exit\n");
        printf("Enter choice: ");
        cin >> choice;
        cin.ignore();
        switch(choice) {
            case 1:
                printf("Enter message: ");
                getline(cin,  message);
                encodeMessageInImage(imageFile, message, outputFile);
                break;
            case 2:
                cout << "The message in the image is: " <<
decodeMessageFromImage(outputFile) << endl;
                break;
            case 3:
                printf("Thank you!\n");
                break;
            default:
                cout << "Invalid choice" << endl;
        }
    } while(choice != 3);
    return 0;
}
```

Steganography is the technique of hiding information within another file (typically images, audio, or text) in such a way that the hidden information is not easily detectable. A common method for image steganography is to hide data in the least significant bits (LSBs) of the image's pixel values, which has minimal visible impact.

Here's a basic implementation of **LSB steganography** in C++ using an image in BMP format.

Key Points:

- **Input image**: A BMP image where data will be hidden.
- **Hidden data**: A string that will be encoded in the image.
- **Output image**: A new BMP image with the hidden data embedded.

Step-by-Step Example:

1. **Read BMP Image**: You can read the BMP file and access pixel data.
2. **Modify Least Significant Bit**: Embed the data by changing the LSB of pixel values.
3. **Write BMP Image**: Save the modified image with the hidden data.

Explanation:

1. **Reading BMP**: The BMP header is first read and copied to the output file.
2. **Encoding**: The `encodeMessageInImage` function modifies the least significant bit (LSB) of each pixel to hide the message bit-by-bit.
3. **Decoding**: The `decodeMessageFromImage` function extracts the message from the LSBs of the pixels until it encounters a null character (`'\0'`), indicating the end of the message.

Important Notes:

- This example works with 24-bit BMP images (each pixel has 3 bytes: Red, Green, and Blue).

- Larger messages require more pixels. Ensure the image has enough pixels to hold the entire message.
- The BMP format is used because it is uncompressed and easy to manipulate. If you're working with other image formats (like PNG, JPEG), you'll need additional libraries for handling these formats.

The function setLSB modifies the **Least Significant Bit (LSB)** of a given byte to a specific value (bit). Let's break it down step by step:

Function Signature:

void setLSB(char &byte, int bit)

- **char &byte**: A reference to the byte (8 bits) whose LSB will be modified.
- **int bit**: The value (either 0 or 1) that will be placed into the LSB of the byte.

Key Operation:

byte = (byte & 0xFE) | (bit & 1);

This line performs two operations:

1. **byte & 0xFE**:
   - **0xFE** in hexadecimal is 11111110 in binary.
   - When you apply a bitwise AND (&) between the byte and 0xFE, the result keeps all the bits of byte the same except the least significant bit (LSB), which is set to 0.
   - For example, if byte is 10110111, applying byte & 0xFE gives 10110110. The LSB (1) becomes 0.
2. **bit & 1**:
   - This ensures that bit is either 0 or 1. Even if a larger integer is passed, applying bit & 1 keeps only the least significant bit of bit.
   - This is important because only 0 or 1 should be placed in the LSB of the byte.
3. **(byte & 0xFE) | (bit & 1)**:
   - After clearing the LSB using byte & 0xFE, the bitwise OR (|) operation sets the LSB to the desired value (bit & 1).
   - If bit is 1, this sets the LSB to 1. If bit is 0, the LSB remains 0.

Example:

Let's look at an example to make this clearer:

Case 1: Setting LSB to 1

- **Initial byte**: 10110110 (in binary)
- **bit = 1**

Operation:

1. **byte & 0xFE**:
   10110110 & 11111110 = 10110110 (the LSB remains 0)
2. **bit & 1**:
   1 & 1 = 1
3. **(byte & 0xFE) | (bit & 1)**:
   10110110 | 00000001 = 10110111

The LSB is now set to 1.

Case 2: Setting LSB to 0

- **Initial byte**: 10110111 (in binary)
- **bit = 0**

Operation:

1. **byte & 0xFE**:
   10110111 & 11111110 = 10110110 (the LSB is cleared to 0)
2. **bit & 1**:
   0 & 1 = 0
3. **(byte & 0xFE) | (bit & 1)**:
   10110110 | 00000000 = 10110110

The LSB is now set to 0.

Summary:

- This function clears the LSB of the byte and replaces it with the specified bit (either 0 or 1).
- It's commonly used in steganography to embed data in the LSBs of image pixels without visibly altering the image.

The function getLSB extracts the **Least Significant Bit (LSB)** of a given byte. Here's a detailed breakdown:

Function Signature:

int getLSB(char byte)

- **char byte**: The input byte (8 bits) from which the LSB is to be extracted.
- **Return type**: The function returns an integer (int), which will either be 0 or 1 (the value of the LSB).

Key Operation:

return byte & 1;

This line performs a **bitwise AND** operation between the byte and the binary number 1 (00000001 in binary).

Explanation:

- **Bitwise AND (&)**: The bitwise AND operation compares each bit of the two operands. If both bits are 1, the result is 1; otherwise, it's 0.
- **Binary representation of 1**: 00000001. When we use the bitwise AND between byte and 1, we are effectively isolating the **least significant bit** of the byte.

Example:

Let's go through a couple of examples to see how this works.

Case 1: When the LSB is 1

- **Input byte**: 10110101 (in binary)

Operation:

10110101 & 00000001 = 00000001

So, the LSB is 1, and the function will return 1.

Case 2: When the LSB is 0

- **Input byte**: 10110100 (in binary)

Operation:

10110100 & 00000001 = 00000000

In this case, the LSB is 0, so the function will return 0.

Summary:

- **Purpose**: The function extracts the least significant bit from a byte.
- **Operation**: It uses the bitwise AND operation between the byte and 1 to isolate the LSB.
- **Return Value**: It returns 1 if the LSB is 1, or 0 if the LSB is 0.

This is useful in steganography when retrieving hidden data from image pixel values, as data is often hidden in the least significant bit.

The function encodeMessageInImage embeds a secret message into the **Least Significant Bits (LSB)** of the pixels in a BMP image. Here's a breakdown of how it works, step-by-step:

Function Signature:

void encodeMessageInImage(const string &imageFile, const string &message, const string &outputFile)

- **imageFile**: The input BMP file where the message will be hidden.
- **message**: The message (as a string) to be hidden in the image.

- **outputFile**: The output BMP file with the hidden message.

Key Operations:

1. Open the Files:

```
ifstream image(imageFile, ios::binary);  // Open input image in binary mode

ofstream output(outputFile, ios::binary); // Open output image in binary mode


if (!image.is_open() || !output.is_open()) {

   cerr << "Error opening file!" << endl;

   return;

}
```

- The input BMP file (imageFile) is opened for reading, and the output BMP file (outputFile) is opened for writing.
- If either file cannot be opened, an error is printed, and the function returns early.

2. Read and Copy BMP Header:

```
char header[54]; // BMP header is 54 bytes

image.read(header, 54);

output.write(header, 54);
```

- The first 54 bytes of a BMP file represent its header, which includes important information such as file size, image width, height, etc.
- This header is read from the input image and immediately copied to the output file without modification.

3. Convert the Message to Binary:

```
string binaryMessage;

for (char c : message) {

   binaryMessage += bitset<8>(c).to_string(); // Convert each char to binary (8 bits per char)

}
```

- Each character in the message is converted into its binary representation (a sequence of 8 bits) using bitset<8>.
- The binary string representing the entire message is stored in binaryMessage.

4. Add a Null Terminator (End Marker):

```
binaryMessage += "00000000";
```

- To mark the end of the hidden message, a null character (00000000 in binary) is appended to the binary message. This acts as a delimiter to indicate the message's end when decoding.

5. Embedding the Message in Image Pixels:

```
char pixel;

size_t messageIndex = 0;


while (image.get(pixel)) {

   if (messageIndex < binaryMessage.size()) {

      setLSB(pixel, binaryMessage[messageIndex] - '0'); // Modify the LSB of the pixel

      messageIndex++;

   }

   output.put(pixel); // Write the pixel (modified or unmodified) to the output

}
```

- A loop reads each byte (pixel data) from the input image.
- If there is still message data to encode (checked by messageIndex < binaryMessage.size()), the function calls setLSB to replace the LSB of the current pixel with the corresponding bit from binaryMessage.
- After modifying the pixel (or leaving it unchanged if the message has already been fully embedded), the pixel is written to the output file.

6. Close the Files:

```
image.close();

output.close();
```

- Both the input and output files are closed after the message embedding is completed.

7. Print Success Message:

```
cout << "Message encoded in " << outputFile << endl;
```

- Once the process is complete, the function prints a success message indicating the output file where the message is encoded.

How It Works:

- The message is converted into binary (1s and 0s), and each bit is embedded into the LSB of each pixel's byte in the BMP file.
- The BMP format is uncompressed, making it easy to directly modify pixel data without affecting the image's visual quality too much.
- The LSBs of pixels are modified because changing them has a minimal visual effect on the image.
- After embedding the message, the modified image is saved as the output BMP file.

Example Flow:

1. Input: imageFile is input.bmp, message is "Hello", and outputFile is output.bmp.
2. The message "Hello" is converted to binary:

'H' -> 01001000

'e' -> 01100101

'l' -> 01101100

'l' -> 01101100

'o' -> 01101111

The final binary message becomes:

01001000 01100101 01101100 01101100 01101111 00000000

3. These bits are embedded into the least significant bits of the pixel bytes in the BMP file.
4. The modified image is saved to output.bmp.

Conclusion:

This function demonstrates how to hide a message within the least significant bits of an image's pixel data using steganography. The output image will look almost identical to the original, but it will have hidden data embedded within the pixel information.

The function decodeMessageFromImage extracts and decodes a hidden message from an image file by reading the **Least Significant Bits (LSBs)** of the image's pixel data. Here's a step-by-step explanation of how it works:

Function Signature:

```
string decodeMessageFromImage(const string &imageFile)
```

- **imageFile**: The input BMP image file from which the message will be decoded.
- **Return type**: The function returns the decoded message as a string.

Key Operations:

1. Open the Image File:

```
ifstream image(imageFile, ios::binary);


if (!image.is_open()) {
   cerr << "Error opening file!" << endl;
   return "";
}
```

- The BMP image file is opened in binary mode for reading.
- If the file cannot be opened, an error message is printed, and the function returns an empty string.

2. Skip BMP Header:

```
image.seekg(54);
```

- The first 54 bytes of the BMP file contain the header, which includes metadata like file size, dimensions, etc.
- The function skips these 54 bytes to access the actual pixel data, where the hidden message resides.

3. Variables Initialization:

```
string binaryMessage;

char pixel;

string message;
```

- **binaryMessage**: A string that will temporarily hold the binary bits of the encoded message (8 bits at a time).
- **pixel**: A variable to read individual bytes (pixels) from the image.
- **message**: The decoded message will be stored here.

4. Extract Message from LSBs:

```
while (image.get(pixel)) {
  binaryMessage += to_string(getLSB(pixel));


  if (binaryMessage.size() == 8) {
    char decodedChar = bitset<8>(binaryMessage).to_ulong();
    if (decodedChar == '\0') {
      break; // Null character found, end of message
    }
    message += decodedChar;
    binaryMessage.clear();
  }
}
```

- A loop reads each byte (pixel) from the image.
- The **getLSB(pixel)** function extracts the least significant bit (LSB) from the current pixel, and this bit is appended to the binaryMessage string.

- Once **8 bits** (1 byte) are collected in binaryMessage, they are converted to a character using bitset<8>(binaryMessage).to_ulong(), which turns the binary string into a numeric value (representing an ASCII character).
- The decoded character is appended to the message string.
- If the decoded character is a **null character (\0)**, it indicates the end of the hidden message, and the loop breaks.
- The binaryMessage is then cleared to process the next 8 bits.

5. Close the File:

image.close();

- The image file is closed after the message extraction is completed.

6. Return the Decoded Message:

return message;

- The function returns the decoded message string.

How It Works:

- The function reads the LSBs of the image's pixel data, where the message is hidden.
- The LSBs are collected in groups of 8 (forming a byte), which are then converted back into characters.
- The process continues until a null character (\0, binary 00000000) is found, which marks the end of the hidden message.

Example Flow:

1. **Input Image**: The input image contains hidden binary data in the LSBs of its pixels.
2. As the function reads pixel data, it extracts LSBs and builds a binary string. For example:

LSBs: 01001000 01100101 01101100 01101100 01101111 00000000

These binary sequences correspond to the ASCII characters:

'H' 'e' 'l' 'l' 'o'

The null terminator (00000000) signals the end of the message.

3. The function converts these binary sequences into characters and assembles the final message, which would be "Hello".

Conclusion:

This function is the decoding part of LSB steganography. It retrieves hidden data by extracting the least significant bits of each pixel's byte in a BMP file. When the binary data has been read and converted back into a string, it returns the secret message hidden within the image.