PixelStealth

A detailed, line-by-line breakdown of the C++ steganography tool's code

Overview

`main.cpp` is the core file for PixelStealth. It implements a simple command-line tool to hide and extract secret messages from PNG images.

Core Concept: LSB Steganography

Hiding data by changing the **L**east **S**ignificant **B**it (the odd/even bit) of a pixel's color byte. This change is invisible to the human eye.

Section 1: Includes & Setup

```
/*
 * PixelStealth: A Simple C++
Steganography Tool
 * Hides secret text messages in PNG
images using
 * Least Significant Bit (LSB)
steganography.
 */

#include
#include
#include
#include

using namespace std;
```

Standard Libraries

- : Provides cout (for printing to console) and cin (for reading user input).
- : Provides the std::string class for handling filenames and the secret message.
- : Provides std::runtime_error, used for throwing exceptions when something goes wrong (e.g., file not found).
- : Used for numeric_limits, which helps clear the input buffer after a failed read.
- using namespace std;: Avoids having to type std:: before every command like cout or string.

Section 1: Image Library Setup (stb)

```
// Define these before
including stb_image headers
#define
STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define
STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"
```

STB Header-Only Libraries

- stb_image.h & stb_image_write.h are popular single-file C libraries for loading and saving images.
- #define STB_IMAGE_IMPLEMENTATION: This special define tells the stb_image.h header to include the *actual function code* (the implementation) in this file.
- #define STB_IMAGE_WRITE_IMPLEMENTATION: This does the same for the stb_image_write.h header.
- This approach simplifies compilation, as there's no separate `.cpp` or library file to link.

Section 2: Global Variables (Image Data)

```
// --- Global Variables ---
unsigned char* g_pixel_data = nullptr;
int g_width, g_height, g_channels;
long long g_data_size = 0;
```

Why Globals?

These are global so all functions (like `hideBit`, `extractBit`, etc.) can access the loaded image data without passing it as a parameter every time.

- g_pixel_data: A pointer that will store the location of the raw image data in memory after it's loaded by `stb_load`. Initialized to `nullptr`.
- g_width, g_height: Integers that will store the image's dimensions in pixels.
- g_channels: Stores the number of color channels (e.g., 3 for RGB, 4 for RGBA).
- g_data_size: The total number of bytes in the image, calculated as `width * height * channels`.

Section 2: Global Variables (State)

```
long long g_data_index =
0;

const string
END_OF_MESSAGE_TAG =
"||EOF||";
```

State Management

- g_data_index: This is the "cursor" or "index" that tracks our current position.
  As we hide/extract each bit, we increment this index to move to the next byte in the `g_pixel_data` array.
  It's reset to `0` every time a new image is loaded.

- END_OF_MESSAGE_TAG: A unique string of characters appended to the end of every secret message.
- **Why?** When extracting, the program reads bytes one by one. It doesn't know when the message ends. This tag acts as a signal, telling the program "Stop reading, the message is complete."

Section 3: Function Prototypes

```
// --- Function
Prototypes ---
void hideMessage();
void
extractMessage();
void
clearInputBuffer();
bool canHideMore();
void hideBit(int
bit_to_hide);
int extractBit();
void hideByte(char
c);
char extractByte();
```

Forward Declarations

- These lines are **forward declarations**. They tell the C++ compiler the names and "signatures" (inputs/outputs) of functions that will be defined *later* in the file.
- This is necessary because `main()` (defined next) calls functions like `hideMessage()` and `extractMessage()`, which are defined at the *end* of the file.
- `canHideMore()` is declared but isn't actually used in the final program.

Section 4: The main() Function (Part 1/4)

```
int main() {
    int choice = -1;

    while (choice != 0) {
        cout <<
"\n═══════════════════════════════════"
<< endl;
        cout << "PixelStealth: C++
Steganography Tool" << endl;
        cout <<
"═══════════════════════════════════"
<< endl;
        cout << "[1] Hide Message in a
PNG" << endl;
        cout << "[2] Extract Message
from a PNG" << endl;
        cout << "[0] Exit" << endl;
        cout << "\nSelect an option: ";

        cin >> choice;
        // ... more code ...
    }
```

The Main Application Loop

- int main(): The entry point of the program.
- int choice = -1;: Initializes the `choice` variable to a non-zero value to ensure the loop runs at least once.
- while (choice != 0): This is the main application loop. It will keep running, showing the menu over and over, until the user enters `0` to exit.
- cout << ...: A series of print statements that display the user menu.
- cin >> choice;: Pauses the program and waits for the user to type a number and press Enter. The number is stored in the `choice` variable.

Section 4: The main() Function (Part 2/4)

```
        // ... menu ...
        cin >> choice;

        if (cin.fail()) {
            choice = -1;

clearInputBuffer();
        }

        try {
            // ... switch
statement ...
        } catch (const
runtime_error& e) {
            // ... error
handling ...
        }
    }
    return 0;
}
```

Robust Input Handling

- if (cin.fail()): This check is crucial. If the user types "abc" instead of a number, cin enters a "failed state". This `if` block catches that.
- choice = -1;: Resets the choice to ensure the `while (choice != 0)` loop continues.
- clearInputBuffer();: Calls our helper function to fix the `cin` stream and clear out the bad "abc" input, making it ready for the next attempt.
- try { ... }: This begins a "try" block. It tells the program to *try* running the code inside, but be prepared for a runtime_error.

Section 4: The main() Function (Part 3/4)

```
        try {
            switch (choice) {
                case 1:
                    hideMessage();
                    break;
                case 2:
                    extractMessage();
                    break;
                case 0:
                    cout <<
"Exiting..." << endl;
                    break;
                default:
                    cout << "Invalid
option." << endl;
                    break;
            }
        } catch (const runtime_error&
e) {
            cerr << "\n!!! ERROR: " <<
e.what() << " !!!" << endl;
            // ... cleanup ...
```

Switch and Catch

- switch (choice): This efficiently executes code based on the user's `choice`.
- case 1: hideMessage();: If `choice` is 1, call the `hideMessage` function.
- case 2: extractMessage();: If `choice` is 2, call the `extractMessage` function.
- catch (const runtime_error& e): If any function inside the `try` block (like `hideMessage`) "throws" an error, the code immediately jumps here.
- cerr << e.what(): Prints the error message (e.g., "File not found?") to the console error stream (`cerr`).

Section 4: The main() Function (Part 4/4)

```
        } catch (const runtime_error& e)
{
            cerr << "\n!!! ERROR: " <<
e.what() << " !!!" << endl;

            if (g_pixel_data) {

stbi_image_free(g_pixel_data);
                g_pixel_data = nullptr;
            }
        }

        if (choice != 0) {
            cout << "\nPress Enter to
continue...";
            clearInputBuffer();
            cin.get();
        }
    }
}
```

Error & Loop Cleanup

- if (g_pixel_data): This check inside the `catch` block is critical. If an error happened *after* an image was loaded, this `if` block runs.
- stbi_image_free(g_pixel_data): This **frees the memory** allocated for the image, preventing a memory leak.
- g_pixel_data = nullptr;: Resets the global pointer so we don't try to free it again.
- if (choice != 0): If the user didn't choose to exit, this pauses the program so they can read the success/error message before the menu loops.
- cin.get();: Waits for the user to press Enter.

Section 5: Helper: clearInputBuffer()

```
void clearInputBuffer() {
    cin.clear();

cin.ignore(numeric_limits::max(),
'\n');
}
```

Fixing `cin`

This function is called after a failed `cin` or before a `getline` to prevent input errors.

- cin.clear();: Resets all error flags on the `cin` stream. If `cin` failed (e.g., user typed "abc" for a number), this function makes it usable again.
- cin.ignore(...): This flushes (discards) all characters currently in the input buffer.
- It's told to ignore up to the maximum possible number of characters (`numeric_limits::max()`) until it finds and removes a newline (`\n`) character.
- This ensures the next `cin` or `getline` starts with a clean slate.

Section 6: Core LSB Logic

The heart of the steganography. These functions manipulate individual bits and bytes.

Core Logic: hideBit()

```
void hideBit(int bit_to_hide) {
    if (g_data_index >=
g_data_size) {
        throw runtime_error("Out
of space!");
    }

    unsigned char& pixel_byte =
g_pixel_data[g_data_index];
    int current_lsb = pixel_byte %
2;

    if (bit_to_hide == 1) {
        if (current_lsb == 0) {
            pixel_byte++; // Make
odd
        }
    } else { // bit_to_hide == 0
        if (current_lsb == 1) {
            pixel_byte--; // Make
even
        }
```

Hiding a Single Bit

- if (g_data_index >= g_data_size): A safety check. Throws an error if the message is too long for the image.
- unsigned char& pixel_byte: Gets a *reference* to the current byte in the image data. The `&` is vital—it means we are modifying the *original* data, not a copy.
- int current_lsb = pixel_byte % 2;: Gets the current LSB. `(number % 2)` is `0` if even, `1` if odd.

- if (bit_to_hide == 1): If we want to hide a '1' (make it odd)...
  - ...and it's currently '0' (even), add 1 (e.g., 254 -> 255).
- else: If we want to hide a '0' (make it even)...
  - ...and it's currently '1' (odd), subtract 1 (e.g., 255 -> 254).
- g_data_index++;: Moves the "cursor" to the next byte, ready for the next bit.

Core Logic: extractBit()

```
int extractBit() {
    if (g_data_index >=
g_data_size) {
        throw
runtime_error("Unexpected end
of file.");
    }

    int bit =
g_pixel_data[g_data_index] % 2;
    g_data_index++;
    return bit;
}
```

Extracting a Single Bit

- if (g_data_index >= g_data_size): Error check. If we run out of image data before finding the `END_OF_MESSAGE_TAG`, the file is corrupt or not a stego-image.
- int bit = g_pixel_data[...] % 2;: The core extraction. It reads the current byte, and the modulo operator (`% 2`) gets its LSB (0 or 1).
- g_data_index++;: Moves the "cursor" to the next byte.
- return bit;: Returns the hidden bit.

Core Logic: hideByte()

```
void hideByte(char c) {
    for (int i = 0; i < 8; ++i)
{
        int bit = (c >> i) & 1;
        hideBit(bit);
    }
}
```

Hiding a Full Character

- This function hides one `char` (1 byte) by calling `hideBit()` 8 times.
- for (int i = 0; i < 8; ++i): Loops 8 times, once for each bit in the character `c`.
- int bit = (c >> i) & 1;: This is the bit extraction logic.
  - (c >> i): The **Bitwise Right Shift** operator. It moves the bits of `c` to the right by `i` places.
  - & 1: The **Bitwise AND** operator. It isolates the *very last* bit.
  - **Example (i=0):** `(01000001 >> 0) & 1` -> `01000001 & 1` -> `1`
  - **Example (i=1):** `(01000001 >> 1) & 1` -> `00100000 & 1` -> `0`
- hideBit(bit);: Hides the extracted bit (0 or 1) in the image.

Core Logic: extractByte()

```
char extractByte() {
    char c = 0;
    for (int i = 0; i < 8; ++i)
{
        int bit = extractBit();
        if (bit == 1) {
            c = c | (1 << i);
        }
    }
    return c;
}
```

Rebuilding a Character

- This function reads 8 bits from the image to rebuild a single `char`.
- char c = 0;: Initializes an empty character (all bits set to `00000000`).
- for (int i = 0; i < 8; ++i): Loops 8 times to get 8 bits.
- int bit = extractBit();: Gets the next hidden bit (0 or 1).
- if (bit == 1): If the hidden bit was a '1'...
- c = c | (1 << i);: ...we "set" that bit in our character `c`.
  - (1 << i): **Bitwise Left Shift**. Creates a "bitmask" (e.g., `00000001`, `00000010`, `00000100`...).
  - c | ...: **Bitwise OR**. Merges the bitmask into `c`.
- return c;: After 8 bits, `c` is fully reassembled (e.g., `01000001`, which is 'A').

Section 7: Main Application

Tying all the logic together in `hideMessage()` and `extractMessage()`.

App Function: hideMessage() (Part 1/3)

```
void hideMessage() {
    string inputFile,
outputFile, message;

    cout << "\nEnter source
image file ... : ";
    clearInputBuffer();
    getline(cin, inputFile);

    cout << "Enter output file
name ... : ";
    getline(cin, outputFile);

    cout << "Enter secret
message: ";
    getline(cin, message);

    if (message.empty()) {
        throw
runtime_error("Message cannot
be empty.");
    }
```

Getting User Input

- Declares strings to hold the filenames and message.
- clearInputBuffer();: Critically, this is called *before* each `getline`. This clears the leftover newline from the `cin >> choice` in `main()`, allowing `getline` to work correctly.
- getline(cin, inputFile);: Reads a whole line of text (including spaces) from the user into the `inputFile` string.
- if (message.empty()): A quick validation check to ensure the user actually entered a message to hide.

App Function: hideMessage() (Part 2/3)

```
    // ... (get input) ...
    cout << "Loading image..."
<< endl;
    g_pixel_data =
stbi_load(inputFile.c_str(),

&g_width, &g_height,
&g_channels, 0);
    if (!g_pixel_data) {
        throw
runtime_error("Failed to load
input image.");
    }

    g_data_size = g_width *
g_height * g_channels;
    g_data_index = 0;

    message +=
END_OF_MESSAGE_TAG;

    if (message.length() * 8 >
```

Loading & Capacity Check

- stbi_load(...): Calls the STB library function to load the image. It fills our global variables (`g_pixel_data`, `g_width`, etc.) directly.
- if (!g_pixel_data): If loading fails, `stbi_load` returns `nullptr`. This check catches the error and throws an exception.
- g_data_size = ...: Calculates the total bytes available for hiding.
- g_data_index = 0;: Resets the "cursor" to the start of the image.
- message += END_OF_MESSAGE_TAG;: Appends the stop tag to the message.
- if (message.length() * 8 ...: This is the **capacity check**. It multiplies the total characters (message + tag) by 8 (bits per char) and checks if this *total bits needed* is greater than the *total bytes available*.
- If it is, we free the memory and throw an error.

App Function: hideMessage() (Part 3/3)

```
    // ... (capacity check) ...
    cout << "Hiding message..."
<< endl;
    for (char c : message) {
        hideByte(c);
    }

    cout << "Saving new
image..." << endl;
    if
(stbi_write_png(outputFile.c_str(
g_width, g_height,

g_channels, g_pixel_data,
                        g_width
* g_channels) == 0)
    {

stbi_image_free(g_pixel_data);
        g_pixel_data = nullptr;
        throw
runtime_error("Failed to write
```

Hiding, Saving, & Cleanup

- for (char c : message): A range-based `for` loop that iterates through every character `c` in the `message` string (which now includes the tag).
- hideByte(c);: Calls our core logic function to hide each character.
- stbi_write_png(...): Calls the STB library function to save the *modified* `g_pixel_data` to the `outputFile`. The last argument (`g_width * g_channels`) is the "stride" or length of one row in bytes.
- if (... == 0): `stbi_write_png` returns 0 on failure. We check this, clean up memory, and throw an error if it fails.
- stbi_image_free(g_pixel_data);: **Crucial cleanup step.** This frees the memory allocated by `stbi_load`. Always done on success or failure.

App Function: extractMessage() (Part 1/3)

```
void extractMessage() {
    string inputFile;
    string extractedMessage = "";

    cout << "\nEnter stego-image file ... :
";
    clearInputBuffer();
    getline(cin, inputFile);

    cout << "Loading image ... " << endl;
    g_pixel_data =
stbi_load(inputFile.c_str(),
                          &g_width,
&g_height, &g_channels, 0);
    if (!g_pixel_data) {
        throw runtime_error("Failed to load
input image.");
    }

    g_data_size = g_width * g_height *
g_channels;
    g_data_index = 0;
```

Loading the Stego-Image

- string extractedMessage = "";: Initializes an empty string to store the characters we find.
- clearInputBuffer(); / getline(...): Same as `hideMessage`, gets the input file from the user.
- stbi_load(...): Loads the potentially modified image into memory.
- if (!g_pixel_data): Throws an error if the file can't be loaded.
- g_data_size = ... / g_data_index = 0;: Sets up the global size and resets the "cursor" to the beginning of the image data.

App Function: extractMessage() (Part 2/3)

```
    // ... (image loaded) ...
    cout << "Extracting
message..." << endl;

    while (true) {
        char c = extractByte();
        extractedMessage += c;

        if
(extractedMessage.length() >

END_OF_MESSAGE_TAG.length())
        {
            if
(extractedMessage.rfind(END_OF_MESS
                != string::npos)
            {
                break; // Tag
found
            }
        }
    }
```

The Extraction Loop

- while (true): An infinite loop that will run until we manually `break` out of it.
- char c = extractByte();: Calls our core logic function to read the next 8 bits from the image and assemble them into a character.
- extractedMessage += c;: Appends the newly found character to our message string.
- if (extractedMessage.length() > ...): A small optimization. We don't bother searching for the tag until our string is at least as long as the tag itself.
- extractedMessage.rfind(...): This searches the `extractedMessage` string for the `END_OF_MESSAGE_TAG`. `rfind` is used to check if the string *ends with* the tag.
- != string::npos: `rfind` returns `string::npos` (a special constant) if the tag is *not* found. If it *is* found, this condition is true.
- break;: Exits the `while(true)` loop.

App Function: extractMessage() (Part 3/3)

```
    // ... (loop finished) ...

stbi_image_free(g_pixel_data);
    g_pixel_data = nullptr;

    size_t tag_pos =
extractedMessage.rfind(

END_OF_MESSAGE_TAG);
    string cleanMessage =
extractedMessage.substr(

0, tag_pos);

    cout << "\n--- SECRET
MESSAGE FOUND ---" << endl;
    cout << cleanMessage <<
endl;
    cout << "-----------------
-------------" << endl;
}
```

Cleanup & Display

- stbi_image_free(g_pixel_data);: Frees the image memory, preventing a leak.
- size_t tag_pos = ...: Finds the starting position of the `END_OF_MESSAGE_TAG` within the extracted string.
- string cleanMessage = ...: Creates a new string by taking a "substring" of the `extractedMessage`. It starts at the beginning (`0`) and goes up to the position of the tag (`tag_pos`).
- This gives us the final message *without* the `||EOF||` tag attached.
- cout << cleanMessage: Prints the final secret message to the console.

Unused Function: canHideMore()

```
bool canHideMore() {
    return (g_data_index +
16) < g_data_size;
}
```

Declared but not Used

- This function was declared in the prototypes but is never called in `main.cpp`.
- Its purpose *would* have been to check if there was enough space left to hide at least a couple more characters (16 bits/bytes).
- The more robust check inside `hideMessage()` (`message.length() * 8 > g_data_size`) makes this function redundant, which is likely why it was left unused.

`main.cpp` Summary

This single file contains the entire application logic, from the user interface and error handling to the low-level bit manipulation, all tied together with the `stb` libraries for image I/O.