I. Overview

1. Folders:

Gồm 3 thư mục: Source, Document và Release.

- Source: Chứa toàn bộ mã nguồn của bài tập
- Release: Chứa file chương trình 21120566.exe
- Document: Chứa file báo cáo 21120566.pdf

2. Code:

Các biến được dùng đúng tên như trong trong slide. Trong các functions còn có thêm 2 prefix là "**src**" và "**dst**" để dễ dàng phân biệt biến gốc và các biến đích. Ngoài ra, các biến cũng được đặt tên theo quy tắc '**camelCase**'.

Ngoài các struct chuẩn theo slide, trong code còn có một struct Pixel dùng để lưu trữ mảng pixel mà mỗi phần tử là 4 màu alpha, blue, green và red. Lí do các màu có kiểu dữ liệu là unsigned char là vì các màu phải không có dấu, nếu không thì màu ảnh sẽ bị lỗi khi tính trung bình cộng. Các vùng sáng là các vùng bị lỗi khi tính trung bình cộng vì giá trị trả về sẽ rất nhỏ nếu tính trung bình cộng của các số có số âm trong đó.

ảnh gốc → ảnh chuyển sang 8-bit → ảnh resize

Em đã băn khoăn rất muốn dùng chỉ 3 màu là blue, green và red cho struct Pixel, vì nếu chỉ có 3 màu, code sẽ sạch và đẹp hơn rất nhiều, tránh tình trạng có nhiều câu điều kiện. Nhưng với struct không có kênh màu alpha thì khi resize ảnh 32-bit, ảnh sẽ bị ám màu như bên dưới (khi chuyển sang 8-bit, ảnh cũng bị ám màu nhưng rất khó để nhận ra).

ảnh gốc → ảnh dùng 3 màu → ảnh dùng 4 màu

Ngoài ra, để đảm bảo không có bất kì lỗi rò rỉ bộ nhớ nào, chương trình đã được check memory leak bằng thư viện **CRT** (C Run-time Library) của **Visual Studio debugger** trước khi release.

II. Functions

Gồm 13 hàm bao gồm: 7 **hàm trả về** và 6 **hàm thông thường** (bao gồm hàm main).

1. Hàm **trả về tổng byte** của ảnh:

```
int getSize(int height, int width, int padding, int bpp);
```

Cách tính tổng byte của ảnh:

```
Tổng byte  = {số pixel} * {số byte mỗi pixel} + {tổng số padding byte}

           = {chiều cao * chiều rộng} * {số bit mỗi byte / 8} + {chiều cao * số padding byte mỗi dòng}

           = height * width * bpp / 8 + height * padding
```

2. Hàm **trả về số padding byte** trên một dòng của ảnh:

```
int getPadding(int width, int bpp);
```

Số padding byte trên một dòng là số byte để bù phần byte bị thiếu để số byte trên dòng chia hết cho 4:

- Nếu chia hết cho 4: Số padding byte sẽ bằng 0
- Nếu không chia hết cho 4: Cách tính số padding byte trên dòng như sau:

```
Số padding byte = 4 - {số byte mỗi dòng} % 4

                = 4 - {chiều rộng * số byte mỗi pixel / 8} % 4

                = 4 - (width * bpp / 8) % 4
```

3. Hàm **trả về vị trí** index trong mảng một chiều tại toạ độ (x, y):

```
int getIndex(int width, int x, int y);
```

Vì trong code chỉ dùng mảng một chiều nên ta cần hàm chuyển vị trí trong mảng hai chiều thành vị trí trong mảng một chiều.

Cách tính vị trí trong mảng một chiều:

```
Vị trí = {toạ độ y} * {chiều rộng} + {toạ độ x}

       =  y * width + x
```

4. Hàm **trả về giá trị trung bình** của một pixel:

```
Pixel getAverage(Pixel srcPixel, Pixel dstPixel);
```

Để giữ code sạch và gọn, ta cũng cần một hàm tính giá trị trung bình giữa pixel cũ và pixel mới:

Cách tính giá trị trung bình giữa hai pixel:

```
Giá trị alpha của pixel mới = ({giá trị alpha của pixel cũ} + {giá trị alpha của pixel mới}) / 2
                            = (dstPixel.A + srcPixel.A) / 2
Giá trị blue của pixel mới  = ({giá trị blue của pixel cũ}  + {giá trị blue của pixel mới})  / 2
                            = (dstPixel.B + srcPixel.B) / 2
Giá trị green của pixel mới = ({giá trị green của pixel cũ} + {giá trị green của pixel mới}) / 2
                            = (dstPixel.G + srcPixel.G) / 2
Giá trị red của pixel mới   = ({giá trị red của pixel cũ}   + {giá trị red của pixel mới})   / 2
                            = (dstPixel.R + srcPixel.R) / 2
```

5. Hàm **đọc file input**:

```
void readBMP(const char* filename, BMP& bmp);
```

Hàm sẽ đọc file input và lưu nó vào một biến có kiểu dữ liệu BMP, bao gồm các bước:

- Mở file input dưới dạng binary
- Xuất lỗi nếu file input không tồn tại và kết thúc chương trình
- Đọc **Header** của file input
- Xuất lỗi nếu ảnh không là ảnh BMP và kết thúc chương trình
- Đọc **DIB** của file input
- Nếu **DIB** lớn hơn 40 bytes: Đọc phần dư của **DIB**

  Vì một số ảnh có phần dư lớn hơn 40 bytes nên ta cần đọc nó và lưu vào **pDIBReserved** để đảm bảo chương trình hoạt động không có bất cứ lỗi nào.
- Nếu không: gán con trỏ **pDIBReserved** cho NULL
- Tính size ảnh của file input

  Vì một số ảnh có size ảnh bị sai nên ta cần tính lại và lưu vào **imageSize** để đảm bảo chương trình hoạt động không có bất cứ lỗi nào.
- Cấp phát vùng nhớ lưu dữ liệu ảnh của file input
- Đọc dữ liệu ảnh của file input
- Đóng file input

6. Hàm **xuất file output**:

```
void exportBMP(const char* filename, BMP bmp);
```

Hàm sẽ đọc một biến có kiểu dữ liệu BMP và lưu nó vào file output, bao gồm các bước:

- Mở/tạo file output dưới dạng binary
- Viết **Header** và **DIB** của file output
- Nếu **DIB** lớn hơn 40 bytes: Viết phần dư của **DIB**

  Vì một số ảnh có phần dư lớn hơn 40 bytes nên ta cần đọc nó và lưu vào **pDIBReserved** để đảm bảo chương trình hoạt động không có bất cứ lỗi nào.
- Nếu file output là ảnh 8-bit: Tạo và viết bảng màu

  Vì với ảnh BMP 8-bit, ta cần phải lưu bảng màu 1024 bytes vào sau phần **DIB**, mỗi byte là một màu blue, green hoặc red nên ta sẽ có 256 cường độ khác nhau (trừ màu alpha luôn bằng 255). Đây được gọi là "Lượng Tử Hoá Màu" (Color Quantization).
- Viết dữ liệu ảnh của file output
- Đóng file output

7. Hàm **trả về mảng pixel** từ mảng màu của ảnh 24-bit/32-bit:

```
Pixel* convertToPixelArray(BMP bmp);
```

Hàm sẽ đọc một biến có kiểu dữ liệu BMP 24-bit/32-bit và chuyển mảng màu **pImageData** sang mảng pixel **newPixel**, bao gồm các bước:

- Gán lại các giá trị
- Cấp phát mảng pixel mới
- **Thuật toán (1): Chuyển từ mảng màu sang mảng pixel**

  Đọc ở **section III**.
- Trả về mảng pixel

8. Hàm **trả về mảng pixel đã resize từ mảng pixel gốc**:

```
Pixel* resizePixelArray(Pixel* srcPixel, int height, int width, int scale);
```

Hàm sẽ đọc một mảng pixel, thu nhỏ nó với các giá trị **height**, **width**, hệ số **scale** cho trước và chuyển sang mảng pixel **newPixel**, bao gồm các bước:

- Gán lại các giá trị, làm tròn trên cho **height** và **width** của ảnh mới

  Vì khi ta thu nhỏ ảnh, nếu ảnh không đều thì ta vẫn sẽ lấy luôn phần dư để tính toán, vì vậy kích thước ảnh mới sẽ bằng làm tròn trên của kích thước ảnh cũ chia cho hệ số **scale**.
- Cấp phát mảng pixel mới
- **Thuật toán (2): Resize mảng pixel**

  Đọc ở **section III**.
- Giải phóng con trỏ mảng pixel
- Trả về mảng pixel mới

9. Hàm **trả về mảng màu** của ảnh 8-bit đã resize từ mảng màu gốc:

```
char* resize8BitArray(char* srcArray, int height, int width, int scale);
```

Hàm sẽ đọc một mảng màu, thu nhỏ nó với các giá trị **height**, **width**, hệ số **scale** cho trước và chuyển sang mảng màu **newArray**, bao gồm các bước:

- Gán lại các giá trị, làm tròn trên cho **height** và **width** của ảnh mới

  Vì khi ta thu nhỏ ảnh, nếu ảnh không đều thì ta vẫn sẽ lấy luôn phần dư để tính toán, vì vậy kích thước ảnh mới sẽ bằng làm tròn trên của kích thước ảnh cũ chia cho hệ số **scale**.

- Cấp phát mảng màu mới
- **Thuật toán (3): Resize** mảng màu

  Đọc ở **section III**.

- Trả về mảng màu mới

10. Hàm **chuyển** ảnh sang **8-bit**:

```
void convertTo8Bit(BMP srcBMP, BMP& dstBMP, Pixel* srcPixel);
```

Hàm sẽ đọc một biến có kiểu dữ liệu BMP 24-bit/32-bit và chuyển mảng màu **pImageData** sang mảng pixel **newPixel**, bao gồm các bước:

- Gán lại các giá trị
- Nếu **DIB** lớn hơn 40 bytes: Gán (copy) lại phần dư của **DIB**

  Vì một số ảnh có phần dư lớn hơn 40 bytes nên ta cần gán lại **pDIBReserved** để đảm bảo chương trình hoạt động không có bất cứ lỗi nào.

- Nếu không: Gán con trỏ cho NULL
- Gán lại các giá trị
- Cấp phát dữ liệu ảnh
- **Thuật toán (4): Lấy trung bình cộng của từng pixel**

  Đọc ở **section III**.

- Giải phóng con trỏ mảng pixel.

11. Hàm **resize** ảnh:

```
void resize(BMP srcBMP, BMP& dstBMP, int scale);
```

Hàm sẽ đọc một biến có kiểu dữ liệu BMP và hệ số **scale** rồi chuyển ảnh sang size nhỏ hơn cho một biến khác có kiểu dữ liệu BMP, bao gồm các bước:

- Gán lại các giá trị
- Nếu **DIB** lớn hơn 40 bytes: Gán (copy) lại phần dư của DIB
- Nếu không: gán con trỏ cho NULL
- Gán lại các giá trị, làm tròn trên cho **height** và **width** của ảnh mới

  Vì khi ta thu nhỏ ảnh, nếu ảnh không đều thì ta vẫn sẽ lấy luôn phần dư để tính toán, vì vậy kích thước ảnh mới sẽ bằng làm tròn trên của kích thước ảnh cũ chia cho hệ số `scale`.
- Cấp phát dữ liệu ảnh
- Nếu ảnh là 8-bit:
  - Cấp phát mảng màu mới
  - **Thuật toán (5): Gán lại dữ liệu ảnh cho ảnh từ mảng màu**
  - Giải phóng con trỏ mảng màu
- Nếu không:
  - Cấp phát mảng pixel mới
  - **Thuật toán (6): Gán lại dữ liệu ảnh cho ảnh từ mảng pixel**
  - Giải phóng con trỏ mảng pixel

12. Hàm **in thông tin** ảnh:

```
void outputInfomations(BMP bmp);
```

Hàm sẽ in ra các thông tin của một biến có kiểu dữ liệu BMP.

13. Hàm **báo lỗi** invalid nếu command line argument không hợp lệ:

```
void invalid();
```

Hàm sẽ báo lỗi và kết thúc chương trình.

14. Hàm **main**:

```
int main(int argc, char* argv[]);
```

Hàm sẽ đọc các đối số dòng lệnh để để nhận giá trị và kiểm tra các câu điều kiện, bao gồm các bước:

- Ignore `argc` để tránh lỗi '`unused parameter`'
- Nếu một trong 3 **argument** đầu không tồn tại: Báo lỗi
- Lưu con trỏ đường dẫn file input và file output của **command line argument**
- Khởi tạo biến BMP cho file input và đọc file input
- Nếu ảnh không phải 8-bit/24-bit/32-bit: Báo lỗi
- Nếu `argv[2]` là "`-conv`":
  - Nếu ảnh là 8-bit: Báo lỗi
  - Nếu không:
    - Khởi tạo biến BMP cho file output
    - Chuyển sang 8-bit, xuất file output, in thông báo và thông tin
    - Giải phóng con trỏ (1)
- Hoặc nếu `argv[2]` là "`-zoom`":
  - Khởi tạo biến BMP cho file output và nhận giá trị scale
  - Resize ảnh, xuất file output, in thông báo và thông tin
  - Giải phóng con trỏ (2)
- Nếu không: Báo lỗi
- Giải phóng con trỏ (3)

III. Algorithms:

1. Thuật toán 1: Chuyển từ mảng màu sang mảng pixel

```
int alpha = bmp.dib.bitsPerPixel / 32;
int i = 0;
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        int index = getIndex(width, x, y);
        newPixel[index].A = bmp.pImageData[i];
        i += alpha;
        newPixel[index].B = bmp.pImageData[i++];
        newPixel[index].G = bmp.pImageData[i++];
        newPixel[index].R = bmp.pImageData[i++];
    }
    i += padding;
}
```

Biến `alpha` là một biến dùng để kiểm tra xem ảnh BMP có phải là 32-bit hay không bằng cách đem chia *`bitsPerPixel`* cho `32`. Nếu ảnh là 32-bit thì `alpha` sẽ bằng `1`, ngược lại sẽ bằng `0` (đối với ảnh 8-bit/24-bit/32-bit).

Biến `i` là một biến dùng để giữ và di chuyển vị trí của con trỏ *`pImageData`* lí do là để cho code dễ nhìn và thuận tiện hơn cho việc debug.

Thuật toán sẽ gồm 2 vòng lặp. Biến `y` sẽ chạy dọc theo chiều cao ảnh và biến `x` sẽ chạy dọc theo chiều rộng ảnh, nhờ đó `index` có thể chạy hết tất cả pixel trong ảnh input (từ `0` đến trước `height*width`):

Trong một `index`, **giả sử ta đang ở vị trí đầu tiên là** `index 0`, màu *A* sẽ nhận giá trị từ *`pImageData`*`[0]`, sau đó cho `i = i + alpha`:

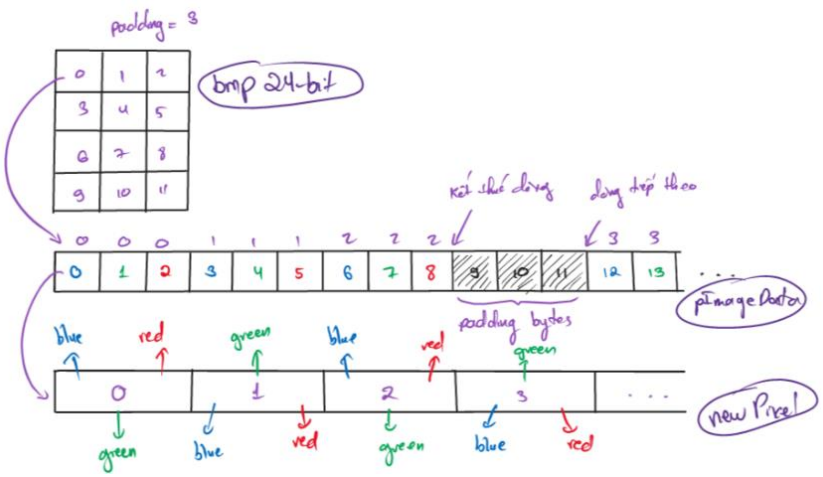
- Nếu `alpha = 0` (ảnh 24-bit) thì `i = 0`, do đó màu *B* sẽ **nhận lại** giá trị từ *`pImageData`*`[0]`, màu *G* và *R* sẽ nhận giá trị từ *`pImageData`*`[1]` và *`pImageData`*`[2]`.

- Nếu `alpha = 1` (ảnh 32-bit) thì `i = 1`, do đó màu *B* sẽ nhận giá trị từ *`pImageData`*`[1]`, màu *G* và *R* sẽ nhận giá trị từ *`pImageData`*`[2]` và *`pImageData`*`[3]`.

Kết thúc mỗi dòng ta sẽ cộng `i` thêm một lượng `padding` để bỏ qua phần byte thừa mỗi dòng. Với ảnh 32-bit `padding` sẽ luôn bằng 0.

Tương tự với các vị trí khác, do đó mỗi phần tử của mảng pixel sẽ lưu trữ giá trị các màu: (*A*), *B*, *G*, *R* trong mỗi điểm ảnh.

Mô phỏng cách hoạt động của thuật toán với ảnh 24-bit.



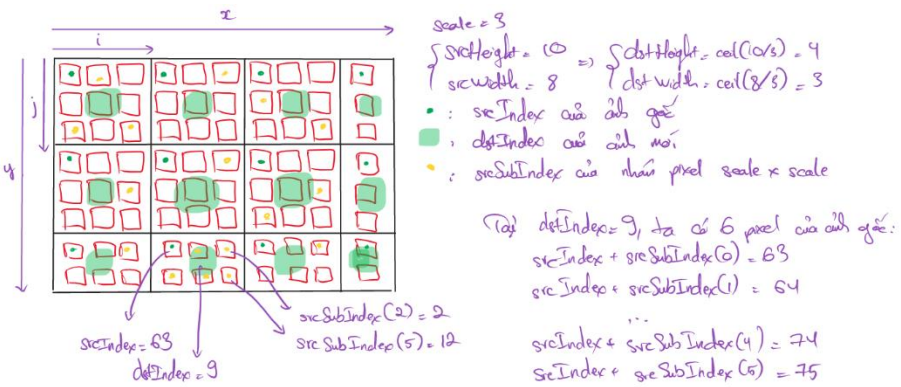
2. Thuật toán 2: Resize mảng pixel

```
for (int y = 0; y < dstHeight; y++) {
    for (int x = 0; x < dstWidth; x++) {
        int srcIndex = getIndex(width, x * scale, y * scale);
        int dstIndex = getIndex(dstWidth, x, y);
        dstPixel[dstIndex] = srcPixel[srcIndex];
        for (int j = 0; j < scale && (j + y * scale < height); j++) {
            for (int i = 0; i < scale && (i + x * scale < width); i++) {
                int srcSubIndex = getIndex(width, i, j);
                dstPixel[dstIndex] = getAverage(srcPixel[srcIndex + srcSubIndex], dstPixel[dstIndex]);
            }
        }
    }
}
```

Biến `dstIndex` là một biến dùng để xác định vị trí của một pixel trong ảnh mới.

Biến `srcIndex` là một biến dùng để xác định vị trí của pixel tại góc trên bên trái trong nhóm pixel có kích thước **`scale` x `scale`** của ảnh gốc.

Biến `srcSubIndex` là vị trí bên trong pixel của nhóm pixel có kích thước **`scale` x `scale`**. Để dễ hiểu hơn, ta nhìn hình minh hoạ bên cạnh.



Thuật toán sẽ gồm 4 vòng lặp:

- 2 vòng lặp ngoài dùng để chạy đến từng pixel **của ảnh mới**: Biến `y` sẽ chạy dọc theo `dstHeight` (chiều cao ảnh mới) và biến `x` sẽ chạy dọc theo `dstWidth` (chiều rộng ảnh mới), nhờ đó `dstIndex` có thể chạy hết tất cả pixel trong ảnh mới (từ `0` đến trước `height*width`).

- 2 vòng lặp trong dùng để chạy đến từng pixel **trong kích thước `scale` x `scale` của ảnh cũ** tại vị trí: Biến `j` và biến `i` sẽ chạy dọc theo `scale`. Nhờ đó `srcIndex * srcSubIndex` có thể trả về vị trí của cả pixel trong nhóm pixel có kích thước **`scale` x `scale`** trong ảnh cũ.

- Trong một `dstIndex`, ta sẽ cho tiếp một vòng lặp kép và với mỗi vòng lặp kép, ta sẽ tính trung bình cộng của `dstPixel` với `srcIndex + srcSubIndex` tại vị trí đó. Vì vậy ta phải cho `dstPixel[dstIndex] = srcPixel[srcIndex]` trước đó để hàm tính trung bình cộng có thể hoạt động với lần lặp đầu tiên (đó là lí do vì sao `srcIndex` là có vị trí ở góc trên bên trái).

Vì vậy, mỗi khi đến một `dstIndex` mới của ảnh mới, ta sẽ lập tức tính được trung bình cộng của các pixel tại đó và gán vào mảng pixel.

3. Thuật toán 3: Resize mảng màu

```
for (int y = 0; y < dstHeight; y++) {
    for (int x = 0; x < dstWidth; x++) {
        int srcIndex = getIndex(width, x * scale, y * scale);
        int dstIndex = getIndex(dstWidth, x, y);
        dstArray[dstIndex] = srcArray[srcIndex];
        for (int j = 0; j < scale && (j + y < height); j++) {
            for (int i = 0; i < scale && (i + x < width); i++) {
                int srcSubIndex = getIndex(width, i, j);
                dstArray[dstIndex] = (srcArray[srcIndex + srcSubIndex] + dstArray[dstIndex]) / 2;
            }
        }
    }
}
```

Tương tự với **thuật toán 2** nhưng thay vì ta dùng hàm để tính trung bình cộng các pixel, ta chỉ cần cộng pixel mới và pixel cũ lại rồi chia cho 2 và lưu vào mảng màu vì các mảng có kiểu dữ liệu *char*).

4. Thuật toán 4: Lấy trung bình cộng của từng pixel

```
int i = 0;
if (dstBMP.dib.bitsPerPixel == 24) {
    for (int y = 0; y < dstHeight; y++) {
        for (int x = 0; x < dstWidth; x++) {
            int index = getIndex(dstWidth, x, y);
            dstBMP.pImageData[i++] = (srcPixel[index].B + srcPixel[index].G + srcPixel[index].R) / 3;
        }
        i += dstPadding;
    }
}
else {
    for (int y = 0; y < dstHeight; y++) {
        for (int x = 0; x < dstWidth; x++) {
            int index = getIndex(dstWidth, x, y);
            dstBMP.pImageData[i++] = (srcPixel[index].A + srcPixel[index].B + srcPixel[index].G + srcPixel[index].R) / 4;
        }
        i += dstPadding;
    }
}
```

Tương tự với **thuật toán 1**, nhưng thay vì gán từng giá trị mà con trỏ *pImageData* trỏ đến vào từng phần tử của mảng pixel thì ta gán từng trung bình cộng của từng phần tử trong mảng pixel vào từng giá trị mà con trỏ *pImageData* trỏ đến.

Nhưng để đảm bảo ảnh được grayscale một cách chính xác, ta phải chia ra 2 trường hợp là 24-bit và 32-bit để tính trung bình cộng phù hợp, tránh màu *B* (từ *A*) được nhân đôi.

5. Thuận toán 5: Gán lại dữ liệu ảnh cho ảnh từ mảng màu

```
int i = 0;
for (int y = 0; y < dstHeight; y++) {
    for (int x = 0; x < dstWidth; x++) {
        int index = getIndex(dstWidth, x, y);
        dstBMP.pImageData[i++] = dstArray[index];
    }
    i += dstPadding;
}
```

Tương tự với **thuật toán 4**, nhưng thay vì lấy trung bình cộng thì ta gán vào luôn vì cả `dstArray` và *`pImageData`* có cùng kiểu dữ liệu.

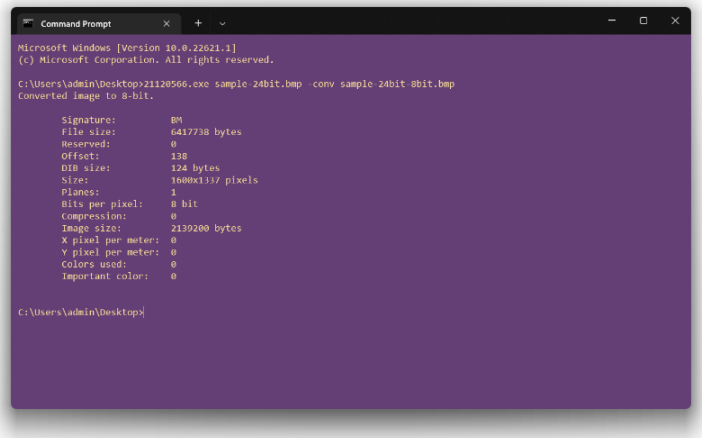
6. Thuật toán 6: Gán lại dữ liệu ảnh cho ảnh từ mảng pixel

```
int i = 0;
int alpha = dstBMP.dib.bitsPerPixel / 32;
for (int y = 0; y < dstHeight; y++) {
    for (int x = 0; x < dstWidth; x++) {
        int index = getIndex(dstWidth, x, y);
        dstBMP.pImageData[i] = dstPixel[index].A;
        i += alpha;
        dstBMP.pImageData[i++] = dstPixel[index].B;
        dstBMP.pImageData[i++] = dstPixel[index].G;
        dstBMP.pImageData[i++] = dstPixel[index].R;
    }
    i += dstPadding;
}
```

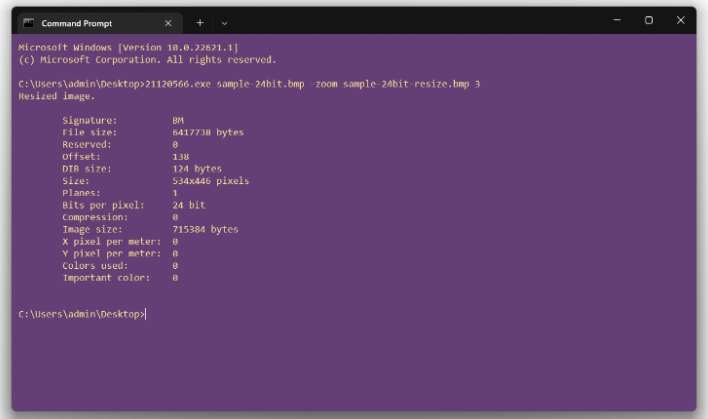
Tương tự với **thuật toán 1**, nhưng thay vì gán từng giá trị mà con trỏ *`pImageData`* trỏ đến vào từng phần tử của mảng pixel thì ta gán từng màu trong từng phần tử trong mảng pixel vào từng giá trị mà con trỏ *`pImageData`* trỏ đến.

IV. Usage

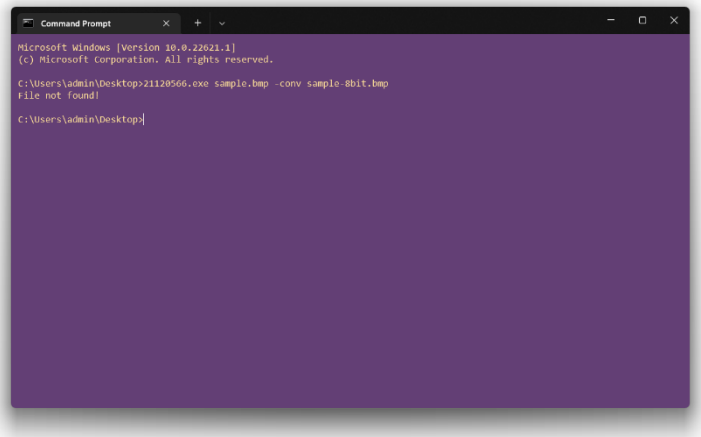
Chuyển ảnh sang 8-bit



Resize ảnh với scale bằng 3



Báo lỗi khi không tìm thấy file



Báo lỗi khi sai cú pháp