```
// morphology.h

#pragma once

#include "image.h"
#include "iostream"

// opening;



image erosion(image im, int struct_matris_dim);

image dilation(image im, int struct_matris_dim);



image opening(image im, int struct_matris_dim);

image closing(image im, int struct_matris_dim);



image edge_detection(image im, int struct_matris_dim);

image & labeling(image &im);

image& regionFilling(image& im);
```

```
// ************************************************************************

// morphology.cpp
#include "morphology.h"



image complement(image im)
{
	int image_size = im.h * im.w;

	unsigned char binary_0 = 0;
	unsigned char binary_1 = 255;

	unsigned char* complement_data = new unsigned char[image_size];
	for (int i = 0; i < image_size; i++)
	{
		complement_data[i] = (im.data[i] == binary_0) ? binary_1 : binary_0; // Complementi
hesapla
		im.data[i] = complement_data[i];
	}

	return im;

}




image erosion(image im, int struct_matris_dim)
{
```

```
int im_column = im.w;

int im_row = im.h;

unsigned char* im_data = im.data;


int m_dim = struct_matris_dim; // 3 ile denendi  // matrisin kaça kaçlık bir kare dimension olacağının atanması

int m_size = m_dim * m_dim; // matris boyutu

int image_size = im_column * im_row; //imajın boyutu


unsigned char* new_data = new unsigned char[image_size]; // erosion uygulanan binary resmin yeni değerlerinin girileceği dizi

for (int i = 0; i < image_size; i++)

{

	new_data[i] = 0;

}

unsigned char* current_region = new unsigned char[m_size]; // resmin işlenecek pikseli ve o pikselin komşularını tutacak matris/dizi

for (int i = 0; i < m_size; i++)

{

	current_region[i] = 0;

}



unsigned char* erosion_matris = new unsigned char[m_size]; // erosion uygulanacak yapısal eleman matrisi/dizisi

for (int i = 0; i < m_size; i++)

{

	erosion_matris[i] = 255;

}



int* ands_result = new int[m_size]; // yapısal eleman ve resmin hedef bölgesinin or işlemi sonuçlarını tutacak matris /dizi
```

```
	for (int i = 0; i < m_size; i++)
	{
		ands_result[i] = 1;
	}
	int and_result = 1; // or işlemi sonucunun and işlemi yapıldıktan sonraki sonucunu tutacak değişken



	// Komşu piksellerin indislerini hesapla
	for (int row = 0; row < im_row; row++) {
		for (int col = 0; col < im_column; col++) {

			// sadece beyaz olanlar üzerinde işlem yap
			if (int(im_data[row * im_column + col]) == 255)
			{
				and_result = 1; // Her piksel için and_result'i sıfırla

				// Kenar piksel kontrolü
				bool isLeftEdge = (col == 0);
				bool isRightEdge = (col == im_column - 1);
				bool isTopEdge = (row == 0);
				bool isBottomEdge = (row == im_row - 1);

				for (int r = 0; r < m_dim; r++) {
					for (int c = 0; c < m_dim; c++) {
						int imgRow = row - 1 + r;
						int imgCol = col - 1 + c;

						// İndislerin sınırlarını kontrol et
						if (imgRow >= 0 && imgRow < im_row && imgCol >= 0 && imgCol < im_column) {
```

```
                    current_region[r * m_dim + c] =
im_data[imgRow * im_column + imgCol];
                }
                else {
                    // Kenar piksel kontrolü
                    if (isLeftEdge && c == 0) { // buralara 0 ata bir
de
                        current_region[r * m_dim + c] = 0; //
Sol kenar pikseli için 1 değeri atanır
                    }
                    else if (isRightEdge && c == m_dim - 1) {
                        current_region[r * m_dim + c] = 0; //
Sağ kenar pikseli için 1 değeri atanır
                    }
                    else if (isTopEdge && r == 0) {
                        current_region[r * m_dim + c] = 0; // !
// Üst kenar pikseli için 1 değeri atanır
                    }
                    else if (isBottomEdge && r == m_dim - 1) {
                        current_region[r * m_dim + c] = 0; //
Alt kenar pikseli için 1 değeri atanır
                    }
                    else {
                        // İndis geçerli değil, dışarıda kalan
bölgeler için isteğe bağlı işlemler yapılabilir
                        // Örneğin, -1 veya farklı bir değer
atanabilir
                        current_region[r * m_dim + c] = 0;
                    }
                }
            }
        }

        for (int i = 0; i < m_size; i++)
```

```
                                { // yapısal elaman or image hedef bölgesi
                                        ands_result[i] = int(current_region[i]) &
int(erosion_matris[i]);
                                }
                                for (int i = 0; i < m_size; i++)
                                { // erosion sonucu değer
                                        and_result = and_result & ands_result[i];
                                }

                                // eğer erosion sonucu 0 ise hedef pikselin değeri azaltılır
                                if (and_result == 0)
                                {
                                        new_data[row * im_column + col] = 0; // !

                                }
                                else
                                {
                                        new_data[row * im_column + col] = 255;
                                }
                                // değilse aynı kalır
                        }
                }
        }

        image erosion_image;
        erosion_image.h = im_row;
        erosion_image.w = im_column;
        erosion_image.c = im.c;
        erosion_image.data = new unsigned char[image_size];

        for (int i = 0; i < image_size; i++)
        {
```

```
                erosion_image.data[i] = new_data[i];
        }

        delete[] im_data;
        delete[] current_region;
        delete[] erosion_matris;
        delete[] ands_result;
        delete[] new_data;

        return erosion_image;
}

image dilation(image im, int struct_matris_dim)
{
        int im_column = im.w;
        int im_row = im.h;
        unsigned char* im_data = im.data;

        int m_dim = struct_matris_dim; // 3 ile denendi  // matrisin kaça kaçlık bir kare dimension olacağının atanması
        int m_size = m_dim * m_dim; // matris boyutu
        int image_size = im_column * im_row; //imajın boyutu

        unsigned char* new_data = new unsigned char[image_size]; // dilation uygulanan binary resmin yeni değerlerinin girileceği dizi
        for (int i = 0; i < image_size; i++)
        {
                new_data[i] = 0;
        }
        unsigned char* current_region = new unsigned char[m_size]; // resmin işlenecek pikseli ve o pikselin komşularını tutacak matris/dizi
        for (int i = 0; i < m_size; i++)
```

```
{
        current_region[i] = 0;
}
unsigned char* dilation_matris = new unsigned char[m_size]; // dilation uygulanacak yapısal eleman matrisi/dizisi
for (int i = 0; i < m_size; i++)
{
        dilation_matris[i] = 0;
}

int* ors_result = new int[m_size]; // yapısal eleman ve resmin hedef bölgesinin or işlemi sonuçlarını tutacak matris /dizi
for (int i = 0; i < m_size; i++)
{
        ors_result[i] = 0;
}
int or_result = 0; // or işlemi sonucunun and işlemi yapıldıktan sonraki sonucunu tutacak değişken

// Komşu piksellerin indislerini hesapla
for (int row = 0; row < im_row; row++) {
        for (int col = 0; col < im_column; col++) {

                or_result = 0; // Her piksel için and_result'i sıfırla

                // Kenar piksel kontrolü
                bool isLeftEdge = (col == 0);
                bool isRightEdge = (col == im_column - 1);
                bool isTopEdge = (row == 0);
                bool isBottomEdge = (row == im_row - 1);

                for (int r = 0; r < m_dim; r++) {
```

```
                for (int c = 0; c < m_dim; c++) {
                    int imgRow = row - 1 + r;
                    int imgCol = col - 1 + c;

                    // İndislerin sınırlarını kontrol et
                    if (imgRow >= 0 && imgRow < im_row && imgCol >= 0 &&
imgCol < im_column) {
                        current_region[r * m_dim + c] = im_data[imgRow *
im_column + imgCol];
                    }
                    else {
                        // Kenar piksel kontrolü
                        if (isLeftEdge && c == 0) { // buralara 0 ata bir de
                            current_region[r * m_dim + c] = 0; // Sol
kenar pikseli için 1 değeri atanır
                        }
                        else if (isRightEdge && c == m_dim - 1) {
                            current_region[r * m_dim + c] = 0; // Sağ
kenar pikseli için 1 değeri atanır
                        }
                        else if (isTopEdge && r == 0) {
                            current_region[r * m_dim + c] = 0; // ! // Üst
kenar pikseli için 1 değeri atanır
                        }
                        else if (isBottomEdge && r == m_dim - 1) {
                            current_region[r * m_dim + c] = 0; // Alt
kenar pikseli için 1 değeri atanır
                        }
                        else {
                            // İndis geçerli değil, dışarıda kalan bölgeler
için isteğe bağlı işlemler yapılabilir
                            // Örneğin, -1 veya farklı bir değer atanabilir
                            current_region[r * m_dim + c] = 0;
                        }
```

```
                        }
                    }
                }

                for (int i = 0; i < m_size; i++)
                { // yapısal elaman or image hedef bölgesi
                    ors_result[i] = int(current_region[i]) | int(dilation_matris[i]);
                }
                for (int i = 0; i < m_size; i++)
                { // dilation sonucu değer
                    or_result = or_result | ors_result[i];
                }

                // eğer dilation sonucu 0 ise hedef pikselin değeri azaltılır
                if (or_result == 0)
                {
                    new_data[row * im_column + col] = 0; // !
                }
                else
                {
                    new_data[row * im_column + col] = 255;
                }
                // değilse aynı kalır

            }
        }

image dilation_image;
dilation_image.h = im_row;
dilation_image.w = im_column;
dilation_image.c = im.c;
```

```
	dilation_image.data = new unsigned char[image_size];

	for (int i = 0; i < image_size; i++)
	{
		dilation_image.data[i] = new_data[i];
	}

	delete[] im_data;
	delete[] current_region;
	delete[] dilation_matris;
	delete[] ors_result;
	delete[] new_data;

	return dilation_image;
}




image opening(image im, int struct_matris_dim)
{
	image im_erosion;
	im_erosion = erosion(im, struct_matris_dim);

	image im_dilation;
	im_dilation = dilation(im_erosion, struct_matris_dim);

	return im_dilation;
}

image closing(image im, int struct_matris_dim)
```

```
{
	image im_dilation;
	im_dilation = dilation(im, struct_matris_dim);

	image im_erosion;
	im_erosion = erosion(im_dilation, struct_matris_dim);



	return im_erosion;
}




image edge_detection(image im, int struct_matris_dim)
{
	int im_column = im.w;
	int im_row = im.h;
	unsigned char* im_data = im.data;

	int m_dim = struct_matris_dim; // 3 ile denendi  // matrisin kaça kaçlık bir kare dimension olacağının atanması
	int m_size = m_dim * m_dim; // matris boyutu
	int image_size = im_column * im_row; //imajın boyutu

	unsigned char* new_data = new unsigned char[image_size]; // dilation uygulanan binary resmin yeni değerlerinin girileceği dizi
	for (int i = 0; i < image_size; i++)
	{
		new_data[i] = 0;
	}
```

```
	unsigned char* current_region = new unsigned char[m_size]; // resmin işlenecek pikseli ve o pikselin komşularını tutacak matris/dizi

	for (int i = 0; i < m_size; i++)

	{

		current_region[i] = 0;

	}



	unsigned char* dilation_matris = new unsigned char[m_size]; // dilation uygulanacak yapısal eleman matrisi/dizisi

	for (int i = 0; i < m_size; i++)

	{

		dilation_matris[i] = 0;

	}



	int* ors_result = new int[m_size]; // yapısal eleman ve resmin hedef bölgesinin or işlemi sonuçlarını tutacak matris /dizi

	for (int i = 0; i < m_size; i++)

	{

		ors_result[i] = 0;

	}

	int or_result = 0; // or işlemi sonucunun and işlemi yapıldıktan sonraki sonucunu tutacak değişken



	// Komşu piksellerin indislerini hesapla

	for (int row = 0; row < im_row; row++) {

		for (int col = 0; col < im_column; col++) {


			if (int(im_data[row * im_column + col]) == 0)
```

```
{
        or_result = 0; // Her piksel için and_result'i sıfırla

        // Kenar piksel kontrolü
        bool isLeftEdge = (col == 0);
        bool isRightEdge = (col == im_column - 1);
        bool isTopEdge = (row == 0);
        bool isBottomEdge = (row == im_row - 1);

        for (int r = 0; r < m_dim; r++) {
            for (int c = 0; c < m_dim; c++) {
                int imgRow = row - 1 + r;
                int imgCol = col - 1 + c;

                // İndislerin sınırlarını kontrol et
                if (imgRow >= 0 && imgRow < im_row && imgCol >=
0 && imgCol < im_column) {
                    current_region[r * m_dim + c] =
im_data[imgRow * im_column + imgCol];
                }
                else {
                    // Kenar piksel kontrolü
                    if (isLeftEdge && c == 0) { // buralara 0 ata bir
de
                        current_region[r * m_dim + c] = 0; //
Sol kenar pikseli için 1 değeri atanır
                    }
                    else if (isRightEdge && c == m_dim - 1) {
                        current_region[r * m_dim + c] = 0; //
Sağ kenar pikseli için 1 değeri atanır
                    }
                    else if (isTopEdge && r == 0) {
```

```
                                        current_region[r * m_dim + c] = 0; // !
// Üst kenar pikseli için 1 değeri atanır
                                    }
                                    else if (isBottomEdge && r == m_dim - 1) {
                                        current_region[r * m_dim + c] = 0; //
Alt kenar pikseli için 1 değeri atanır
                                    }
                                    else {
                                        // İndis geçerli değil, dışarıda kalan
bölgeler için isteğe bağlı işlemler yapılabilir
                                        // Örneğin, -1 veya farklı bir değer
atanabilir
                                        current_region[r * m_dim + c] = 0;
                                    }
                                }
                            }
                        }

                        for (int i = 0; i < m_size; i++)
                        { // yapısal elaman or image hedef bölgesi
                            ors_result[i] = int(current_region[i]) | int(dilation_matris[i]);
                        }
                        for (int i = 0; i < m_size; i++)
                        { // dilation sonucu değer
                            or_result = or_result | ors_result[i];
                        }

                        // eğer dilation sonucu 0 ise hedef pikselin değeri azaltılır
                        if (or_result == 0)
                        {
                            new_data[row * im_column + col] = 0; // !
                        }
```

```
                                else
                                {
                                        new_data[row * im_column + col] = 255;
                                }
                                // değilse aynı kalır
                        }
                }
        }

        image dilation_image;
        dilation_image.h = im_row;
        dilation_image.w = im_column;
        dilation_image.c = im.c;
        dilation_image.data = new unsigned char[image_size];

        for (int i = 0; i < image_size; i++)
        {
                dilation_image.data[i] = new_data[i];
        }

        delete[] im_data;
        delete[] current_region;
        delete[] dilation_matris;
        delete[] ors_result;
        delete[] new_data;

        return dilation_image;
}
```

```
void connectedComponent(image im, int row, int column, unsigned char label) {
	if (row < 0 || column < 0 || row >= im.h || column >= im.w) {
		return;
	}

	if (im.data[row * im.w + column] != 255) {
		return;
	}

	im.data[row * im.w + column] = label;

	connectedComponent(im, row - 1, column, label);
	connectedComponent(im, row + 1, column, label);
	connectedComponent(im, row, column - 1, label);
	connectedComponent(im, row, column + 1, label);
}

image& labeling(image& im)
{
	int row = im.h;
	int column = im.w;

	int label = 20;


	for (int r = 0; r < row; r++) {
		for (int c = 0; c < column; c++) {
			if (im.data[r*column+c] == 255) {
				connectedComponent(im, r, c, label);
				label = label+10;
```

```
                        }
                }
        }

        return im;
}



// çalışmadı
image& regionFilling(image& im) {
        int row = im.h;
        int column = im.w;

        unsigned char label = 255;

        image filled_image;
        filled_image.h = row;
        filled_image.w = column;
        filled_image.c = 1;

        filled_image.data = new unsigned char[row * column]{ 0 };

        for (int r = 0; r < row; r++) {
                for (int c = 0; c < column; c++) {

                        // imajdaki beyaz noktalar
                        if (im.data[r * column + c] == 255)
                        {

                                // center
```

```
                        if (!(r < 0 || c < 0 || r >= row || c >= column)) {
                                filled_image.data[r * column + c] = label;
                        }
                        // top
                        if (!(r - 1 < 0 || c < 0 || r - 1 >= row || c >= column)) {
                                filled_image.data[(r - 1) * column + c] = label;
                        }
                        // bottom
                        if (!(r + 1 < 0 || c < 0 || r + 1 >= row || c >= column)) {
                                filled_image.data[(r + 1) * column + c] = label;
                        }
                        // left
                        if (!(r < 0 || c - 1 < 0 || r >= row || c - 1 >= column)) {
                                filled_image.data[r * column + (c - 1)] = label;
                        }
                        // right
                        if (!(r < 0 || c + 1 < 0 || r >= row || c + 1 >= column)) {
                                filled_image.data[r * column + (c + 1)] = label;
                        }

                }
        }
    }

    return filled_image;
}
```