중간 - 1~7주 실습 파일 정리

```
#pragma warning(disable:4996) // 입력 함수 보안 해결
#include <stdio.h>
#include <stdlib.h> // 동적 메모리 할당 라이브러리
#include <Windows.h> // 파일 헤더, 인포 헤더, RGB QUAD 헤더 구조체 가져오는 라이브러리
#include <math.h>

void InverseImage(BYTE *Img, BYTE *Out, int W, int H) {
  int ImgSize = W * H;
  for (int i = 0; i < ImgSize; i++) {
    Out[i] = 255-Img[i];
  }
}

void BrightnessAdj(BYTE* Img, BYTE* Out, int W, int H, int Val) {
  int ImgSize = W * H;
  for (int i = 0; i < ImgSize; i++) {
    if (Img[i] + Val > 255) { // 클리핑 처리
      Out[i] = 255;
    }
    else if (Img[i] + Val < 0) {
      Out[i] = 0;
    }
    else Out[i] = Img[i] + Val;
  }
}

void ContrastAdj(BYTE* Img, BYTE* Out, int W, int H, double Val) {
  int ImgSize = W * H;
  for (int i = 0; i < ImgSize; i++) {
    // 대비 처리 할 때는, 조건문을 double 타입으로 처리한다.
    if (Img[i] * Val > 255.0) { // 클리핑 처리
      Out[i] = 255;
    }
    else if (Img[i] * Val < 0.0) {
      Out[i] = 0;
    }
    else Out[i] = (BYTE)(Img[i] * Val); // 마지막 형 변환을 double 로 강제 형변환
  }
}

void ObtainHistogram(BYTE* Img, int* Histo, int W, int H) { // 히스토그램 구하기
  int ImgSize = W * H;
  for (int i = 0; i < ImgSize; i++) {
    Histo[Img[i]]++;
  }
}

void ObtainAHistogram(int* Histo, int* AHisto) { // 누적 히스토그램 구하기
  for (int i = 0; i < 256; i++) {
    for (int j = 0; j <= i; j++) {
      AHisto[i] += Histo[j];
    }
  }

}

void HistogramStretching(BYTE * Img, int * Histo, BYTE * Out, int W, int H) { // 히스토그램 스트레칭
  int ImgSize = W * H;
  BYTE Low, High;
  for (int i = 0; i < 256; i++) {
    if (Histo[i] != 0) {
      Low = i;
      break;
    }
  }
  for (int i = 255; i >= 0; i--) {
    if (Histo[i] != 0) {
      High = i;
      break;
    }
  }
  for (int i = 0; i < ImgSize; i++) {
    Out[i] = (BYTE)((Img[i] - Low) / (double)(High - Low) * 255.0);
  }
}

void HistogramEqualization(BYTE * Img, BYTE * Out, int* AHisto, int *NormSum, int W, int H) { // 평활화 : 히스토그램 고르게 분포
  int ImgSize = W * H;
  int Nt = W * H; int Gmax = 255;
  double Ratio = Gmax / (double)Nt;
```

```
  for (int i = 0; i < 256; i++) {
    NormSum[i] = (BYTE)(Ratio * AHisto[i]); // Ratio-Double, AHisto-Int => BYTE 로 형 변환
  }
  for (int i = 0; i < ImgSize; i++) {
    Out[i] = NormSum[Img[i]];
  }
}

void Binarization(BYTE * Img, BYTE * Out, int W, int H, int T) { // 영상 이진화
  int ImgSize = W * H;
  for (int i = 0; i < ImgSize; i++) {
    if (Img[i] < T) Out[i] = 0;
    else Out[i] = 255;
  }
}

int GonzalezBinThresh(int * Histo) {

  BYTE Low, High;
  BYTE T ,New_T;
  int G1=0, G2=0,m1,m2; // G1 : 임계치 보다 밝은 영역, G2 : 임계치 보다 어두운 영역
  int count1 = 0, count2 = 0;
  for (int i = 0; i < 256; i++) {
    if (Histo[i] != 0) {
      Low = i;
      break;
    }
  }
  for (int i = 255; i >= 0; i--) {
    if (Histo[i] != 0) {
      High = i;
      break;
    }
  }
  T = (Low + High) / 2; // 초기 임계 값
  while (1) {
    for (int i = T + 1; i <= High; i++) {
      G1 += Histo[i] * i; // 밝은 영역의 밝기값 합
      count1 += Histo[i]; // 개수
    }
    for (int i = Low; i <= T; i++) {
      G2 += Histo[i] * i;  // 어두운 영역의 밝기값 합
      count2 += Histo[i]; // 개수
    }
    m1 = G1 / count1; // G1 영역의 평균
    m2 = G2 / count2; // G2 영역의 평균

    New_T = (m1 + m2) / 2; // 새로운 T
    if (abs(New_T - T) < 3) {
      T = New_T;
      break;
    }
    else T = New_T;
    G1 = G2 = count1 = count2 = 0;
  }
  return T; // 임계치 반환
}

void SaveBMPFile(BITMAPFILEHEADER hf, BITMAPINFOHEADER hInfo, RGBQUAD *hRGB, BYTE *Output, int W, int H, const char* FileName) {
  FILE *fp = fopen(FileName, "wb");
  fwrite(&hf, sizeof(BYTE), sizeof(BITMAPFILEHEADER), fp);
  fwrite(&hInfo, sizeof(BYTE), sizeof(BITMAPINFOHEADER), fp);
  fwrite(hRGB, sizeof(RGBQUAD), 256, fp);
  fwrite(Output, sizeof(BYTE), W*H, fp);
  fclose(fp); // 포인터 변수 끊어준다.
}

void AverageConv(BYTE *Img, BYTE *Out , int W, int H) { // 평활화 박스
  double Kernel[3][3] = { 0.11111, 0.11111, 0.11111,
                  0.11111, 0.11111, 0.11111,
                  0.11111, 0.11111, 0.11111 };
  double SumProduct = 0.0;
  for (int i = 1; i < H - 1; i++) { // Y좌표 (행)
    for (int j = 1; j < W - 1; j++) { // X좌표 (열)
      for (int m = -1; m <= 1; m++) { // Kernel 행
        for (int n = -1; n <= 1; n++) { // Kernel 열
          SumProduct += Img[(i + m) * W + (j + n)] * Kernel[m + 1][n + 1];
        }
      }
      Out[i * W + j] = (BYTE)SumProduct;
      SumProduct = 0.0;
    }
  }
}

void GaussAvrConv(BYTE* Img, BYTE* Out, int W, int H) {
  double Kernel[3][3] = {
```

```
    0.0625, 0.125, 0.0625,
    0.125, 0.25, 0.125,
    0.0625, 0.125,0.0625
  };
  double SumProduct = 0.0;
  for (int i = 1; i < H - 1; i++) { // Y좌표 (행)
    for (int j = 1; j < W - 1; j++) { // X좌표 (열)
      for (int m = -1; m <= 1; m++) { // Kernel 행
        for (int n = -1; n <= 1; n++) { // Kernel 열
          SumProduct += Img[(i + m) * W + (j + n)] * Kernel[m + 1][n + 1];
        }
      }
      Out[i * W + j] = (BYTE)SumProduct;
      SumProduct = 0.0;
    }
  }
}

void Prewitt_X_Conv(BYTE* Img, BYTE* Out, int W, int H) { //경계(Edge)가 세로로 검출된다. abs((long)SumProduct)/3
  double Kernel[3][3] = {
    -1.0, 0.0,1.0,
    -1.0, 0.0,1.0,
    -1.0, 0.0,1.0
  };
  double SumProduct = 0.0;
  for (int i = 1; i < H - 1; i++) { // Y좌표 (행)
    for (int j = 1; j < W - 1; j++) { // X좌표 (열)
      for (int m = -1; m <= 1; m++) { // Kernel 행
        for (int n = -1; n <= 1; n++) { // Kernel 열
          SumProduct += Img[(i + m) * W + (j + n)] * Kernel[m + 1][n + 1];
        }
      }
      // 0부터 765 ==> 0부터 255까지로 변경 1. abs (음수일수 있기때문) 2. *1/3 (3으로 나눈다)
      Out[i * W + j] = abs((long)SumProduct)/3; // SumProduct는 실수형태 -> 형변환 해야 절대값 가능
      SumProduct = 0.0;
    }
  }
}
void Prewitt_Y_Conv(BYTE* Img, BYTE* Out, int W, int H) { //경계(Edge)가 세로로 검출된다. abs((long)SumProduct)/3
  double Kernel[3][3] = {
    -1.0,-1.0,-1.0,
     0.0, 0.0, 0.0,
     1.0, 1.0, 1.0,
  };
  double SumProduct = 0.0;
  for (int i = 1; i < H - 1; i++) { // Y좌표 (행)
    for (int j = 1; j < W - 1; j++) { // X좌표 (열)
      for (int m = -1; m <= 1; m++) { // Kernel 행
        for (int n = -1; n <= 1; n++) { // Kernel 열
          SumProduct += Img[(i + m) * W + (j + n)] * Kernel[m + 1][n + 1];
        }
      }
      // 0부터 765 ==> 0부터 255까지로 변경 1. abs (음수일수 있기때문) 2. *1/3 (3으로 나눈다)
      Out[i * W + j] = abs((long)SumProduct) / 3; // SumProduct는 실수형태 -> 형변환 해야 절대값 가능
      SumProduct = 0.0;
    }
  }
}

void Sobel_X_Conv(BYTE* Img, BYTE* Out, int W, int H) { //경계(Edge)가 세로로 검출된다. abs((long)SumProduct)/4
  double Kernel[3][3] = {
    -1.0, 0.0,1.0,
    -2.0, 0.0,2.0,
    -1.0, 0.0,1.0
  };
  double SumProduct = 0.0;
  for (int i = 1; i < H - 1; i++) { // Y좌표 (행)
    for (int j = 1; j < W - 1; j++) { // X좌표 (열)
      for (int m = -1; m <= 1; m++) { // Kernel 행
        for (int n = -1; n <= 1; n++) { // Kernel 열
          SumProduct += Img[(i + m) * W + (j + n)] * Kernel[m + 1][n + 1];
        }
      }
      // 0부터 1025 ==> 0부터 255까지로 변경 1. abs (음수일수 있기때문) 2. *1/4 (3으로 나눈다)
      Out[i * W + j] = abs((long)SumProduct) / 4; // SumProduct는 실수형태 -> 형변환 해야 절대값 가능
      SumProduct = 0.0;
    }
  }
}
void Sobel_Y_Conv(BYTE* Img, BYTE* Out, int W, int H) { //경계(Edge)가 세로로 검출된다. abs((long)SumProduct)/4
  double Kernel[3][3] = {
    -1.0,-2.0,-1.0,
     0.0, 0.0, 0.0,
     1.0, 2.0, 1.0,
  };
  double SumProduct = 0.0;
  for (int i = 1; i < H - 1; i++) { // Y좌표 (행)
```

```
    for (int j = 1; j < W - 1; j++) { // X좌표 (열)
      for (int m = -1; m <= 1; m++) { // Kernel 행
        for (int n = -1; n <= 1; n++) { // Kernel 열
          SumProduct += Img[(i + m) * W + (j + n)] * Kernel[m + 1][n + 1];
        }
      }
      // 0부터 1024 ==> 0부터 255까지로 변경 1. abs (음수일수 있기때문) 2. *1/4 (4으로 나눈다)
      Out[i * W + j] = abs((long)SumProduct) / 4; // SumProduct는 실수형태 -> 형변환 해야 절대값 가능
      SumProduct = 0.0;
    }
  }
}
void Laplace_Conv(BYTE* Img, BYTE* Out, int W, int H) {  // 전방향 경계 설정 abs((long)SumProduct)/8
  double Kernel[3][3] = {
    -1.0,-1.0,-1.0,
    -1.0, 8.0, -1.0,
    -1.0, -1.0, -1.0,
  };
  double SumProduct = 0.0;
  for (int i = 1; i < H - 1; i++) { // Y좌표 (행)
    for (int j = 1; j < W - 1; j++) { // X좌표 (열)
      for (int m = -1; m <= 1; m++) { // Kernel 행
        for (int n = -1; n <= 1; n++) { // Kernel 열
          SumProduct += Img[(i + m) * W + (j + n)] * Kernel[m + 1][n + 1];
        }
      }
      // 0부터 2040 ==> 0부터 255까지로 변경 1. abs (음수일수 있기때문) 2. *1/8 (8으로 나눈다)
      Out[i * W + j] = abs((long)SumProduct) / 8; // SumProduct는 실수형태 -> 형변환 해야 절대값 가능
      SumProduct = 0.0;
    }
  }
}
void Laplace_Conv_DC(BYTE* Img, BYTE* Out, int W, int H) { // 고역통과 필터, 노이즈 강조(증폭) , 노이즈 클리핑 처리
  double Kernel[3][3] = {
    -1.0,-1.0,-1.0,
    -1.0, 9.0, -1.0,
    -1.0, -1.0, -1.0,
  };
  double SumProduct = 0.0;
  for (int i = 1; i < H - 1; i++) { // Y좌표 (행)
    for (int j = 1; j < W - 1; j++) { // X좌표 (열)
      for (int m = -1; m <= 1; m++) { // Kernel 행
        for (int n = -1; n <= 1; n++) { // Kernel 열
          SumProduct += Img[(i + m) * W + (j + n)] * Kernel[m + 1][n + 1];
        }
      }
      // 클리핑 처리
      if (SumProduct > 255.0) Out[i * W + j] = 255;
      else if (SumProduct < 0.0) Out[i * W + j] = 0;
      else Out[i * W + j] = (BYTE)SumProduct;
      SumProduct = 0.0;
    }
  }
}
void swap(BYTE *a, BYTE * b) {
  BYTE temp = *a;
  *a = *b;
  *b = temp;
}

int Median(BYTE *arr, int size) {
  for (int i = 0; i < size - 1; i++) {
    for (int j = i + 1; j < size; j++) {
      if (arr[i] > arr[j]) swap(&arr[i],&arr[j]);
    }
  }
  return arr[4]; // 중간값 리턴
}

int push(short* stackx, short* stacky, int arr_size, short vx, short vy, int* top)
{
  if (*top >= arr_size) return(-1);
  (*top)++;
  stackx[*top] = vx;
  stacky[*top] = vy;
  return(1);
}

int pop(short* stackx, short* stacky, short* vx, short* vy, int* top)
{
  if (*top == 0) return(-1);
  *vx = stackx[*top];
  *vy = stacky[*top];
  (*top)--;
  return(1);
}
```

```
// GlassFire 알고리즘을 이용한 라벨링 함수
// GlassFire 알고리즘을 이용한 라벨링 함수
void m_BlobColoring(BYTE* CutImage, int height, int width)
{
  int i, j, m, n, top, area, Out_Area, index, BlobArea[1000];
  long k;
  short curColor = 0, r, c;
  //   BYTE** CutImage2;
  Out_Area = 1;


  // 스택으로 사용할 메모리 할당
  short* stackx = new short[height * width];
  short* stacky = new short[height * width];
  short* coloring = new short[height * width];

  int arr_size = height * width;

  // 라벨링된 픽셀을 저장하기 위해 메모리 할당

  for (k = 0; k < height * width; k++) coloring[k] = 0;  // 메모리 초기화

  for (i = 0; i < height; i++)
  {
    index = i * width;
    for (j = 0; j < width; j++)
    {
      // 이미 방문한 점이거나 픽셀값이 255가 아니라면 처리 안함
      if (coloring[index + j] != 0 || CutImage[index + j] != 255) continue;
      r = i; c = j; top = 0; area = 1;
      curColor++;

      while (1)
      {
      GRASSFIRE:
        for (m = r - 1; m <= r + 1; m++)
        {
          index = m * width;
          for (n = c - 1; n <= c + 1; n++)
          {
            //관심 픽셀이 영상경계를 벗어나면 처리 안함
            if (m < 0 || m >= height || n < 0 || n >= width) continue;

            if ((int)CutImage[index + n] == 255 && coloring[index + n] == 0)
            {
              coloring[index + n] = curColor; // 현재 라벨로 마크
              if (push(stackx, stacky, arr_size, (short)m, (short)n, &top) == -1) continue;
              r = m; c = n; area++;
              goto GRASSFIRE;
            }
          }
        }
        if (pop(stackx, stacky, &r, &c, &top) == -1) break;
      }
      if (curColor < 1000) BlobArea[curColor] = area;
    }
  }

  float grayGap = 255.0f / (float)curColor; // curColor : 총 Component 개수(약 25개) 255.0f/25 = 10.xxxx
  //=> 10.xx px 간격으로 Component의 밝기값을 Filling 해준다.


  // 라벨링 하는 것 뿐만 아니라 Component 중 가장 큰 것을 찾아서 가장 큰 것만 출력을 해준다.

  // 가장 면적이 넓은 영역을 찾아내기 위함 , BlobArea 는 각 컴퍼넌트들의 면적정보가 저장되어 있다.
  for (i = 1; i <= curColor; i++)
  {
    if (BlobArea[i] >= BlobArea[Out_Area]) Out_Area = i;
  }
  // CutImage 배열 클리어~
  for (k = 0; k < width * height; k++) CutImage[k] = 255; // 메인함수에서 Output을 CutImage로 전달받음. 전부다 255(하얗게)로 채운다.

  // coloring에 저장된 라벨링 결과중 (Out_Area에 저장된) 영역이 가장 큰 것만 CutImage에 저장
  for (k = 0; k < width * height; k++)
  {
    if (coloring[k] == Out_Area) CutImage[k] = 0;  // Size Filtering : 가장 큰 Component만 0(검게)으로 채운다.
    //if (BlobArea[coloring[k]] > 500) CutImage[k] = 0;  // 특정 면적이상 되는 영역만 출력
    //CutImage[k] = (unsigned char)(coloring[k] * grayGap); // 라벨링된 결과를 그대로 모두 출력,
    // coloring 배열은 Index정보가 들어가있음. 컴포넌트 숫자가 들어가있음(동일한 영역인지 판별), 1번영역인지, 2번영역인지
    // BlobArea 배열은 각 컴퍼넌트들의 면적정보가 저장되어 있다
    // 배경이 255인 상태에서 각각의 Component들을 밝기값으로 채워나가는 과정
  }

  delete[] coloring;
  delete[] stackx;
  delete[] stacky;
```

```
}
// 라벨링 후 가장 넓은 영역에 대해서만 뽑아내는 코드 포함

void BinaryImageEdgeDetection(BYTE * bin, BYTE * Out, int W, int H) { // bin 은 이진화된 영상을 의미
  for (int i = 0; i < H; i++) { // 경계 검출 식
    for (int j = 0; j < W; j++) {
      if (bin[i * W + j] == 0) { // 전경화소라면
        // 4주변화소 경계검출
        if (!(bin[(i - 1) * W + j] == 0 && bin[(i + 1) * W + j] == 0 && bin[i * W + (j - 1)] == 0 && bin[i * W + (j + 1)] == 0)) { //경
          Out[i * W + j] = 255;
        }
      }
    }
  }
}

void VerticalFlip(BYTE *Img, int W, int H) {
  for (int i = 0; i < H / 2; i++) {// y
    for (int j = 0; j < W; j++) {// x
      swap(&Img[i*W+j], &Img[(H-1-i)*W+j]);
    }
  }
}

void HorizontalFlip(BYTE* Img, int W, int H) {
  for (int i = 0; i < W / 2; i++) { //x
    for (int j = 0; j < H; j++) { //y
      swap(&Img[j * W + i], &Img[j * W + (W-1-i)]);
    }
  }
}

void Translation(BYTE* Image, BYTE* Output, int W, int H, int Tx, int Ty) {
  Ty = Ty * -1;
  for (int i = 0; i < H; i++) {
    for (int j = 0; j < W; j++) {
      if ((i + Ty < H && i + Ty >= 0) && (j + Tx < W && j + Tx >= 0))
        Output[(i + Ty) * W + (j + Tx)] = Image[i * W + j];
    }
  }
}

void Scaling(BYTE *Image, BYTE *Output, int W, int H, double SF_X, double SF_Y) {
  // 순방향 스케일링 i*W+j
  /*
  int tmp_X, tmp_Y;
  for (int i = 0; i < H; i++) {
    for (int j = 0; j < W; j++) {
      tmp_X = (int)j * SF_X;
      tmp_Y = (int)i * SF_Y;
      if (tmp_Y<H && tmp_X <W)
      Output[tmp_Y*W+tmp_X] = Image[i*W+j];
    }
  }*/

  // 역방향 스케일링 Output에 i*W+j
  int tmp_X, tmp_Y;
  for (int i = 0; i < H; i++) {
    for (int j = 0; j < W; j++) {
      tmp_X = (int)j / SF_X;
      tmp_Y = (int)i / SF_Y;
      if (tmp_Y < H && tmp_X < W)
        Output[i * W + j] = Image[tmp_Y * W + tmp_X];
    }
  }
}

void Rotation(BYTE *Image, BYTE * Output, int W , int H, int Angle) {
  double Radian = Angle * 3.141592 / 180.0;
  int tmp_X, tmp_Y;
  /* 순방향 -> Hole 문제 발생 , Input에 i*W+j
  for (int i = 0; i < H; i++) {
    for (int j = 0; j < W; j++) {
      tmp_X = (int)(cos(Radian)*j - sin(Radian)*i);
      tmp_Y = (int)(sin(Radian)*j + cos(Radian)*i);
      if ((tmp_Y < H && tmp_Y >0) && (tmp_X < W && tmp_X>0))
        Output[tmp_Y * W + tmp_X] = Image[i * W + j];
    }
  }
  */

  // 역방향 : 반시계방향으로 갈 것을 시계방향으로 바꿔준다, 시계방향으로 갈 것을 반시계 방향 행렬로 바꿔준다.
  for (int i = 0; i < H; i++) { // Output에 i*W+j
    for (int j = 0; j < W; j++) {
      tmp_X = (int)(cos(Radian) * j + sin(Radian) * i);
      tmp_Y = (int)(-sin(Radian) * j + cos(Radian) * i);
      if ((tmp_Y < H && tmp_Y >0) && (tmp_X < W && tmp_X>0))
```

```
        Output[i * W + j] = Image[tmp_Y * W + tmp_X];
    }
  }
}

int main()
{
  // 1. 영상 입력
  BITMAPFILEHEADER hf; // BMP 파일헤더 14Bytes
  BITMAPINFOHEADER hInfo; // BMP 인포헤더 40Bytes
  RGBQUAD hRGB[256]; // 팔레트 (256 * 4Bytes)
  FILE* fp;
  fp = fopen("LENNA.bmp", "rb");
  if (fp == NULL) {
    printf("File Not Found!\n");
    return -1;
  } // 파일 포인터가 NULL이면 File Not Found를 출력한다.
  fread(&hf, sizeof(BITMAPFILEHEADER), 1, fp); //라인이 끝나면 fp는 Info 헤더의 처음을 가리킨다.
  fread(&hInfo, sizeof(BITMAPINFOHEADER), 1, fp); //라인이 끝나면 fp는 RGBQUAD 헤더의 처음을 가리킨다.
  fread(hRGB, sizeof(RGBQUAD), 256, fp); // 배열의 이름이 곧 주소이기 때문에 주소(&)를 붙이지 않는다. 4바이트씩 256번 읽는다. 라인이 끝나면 fp는 영상의 화소

  int W = hInfo.biWidth; int H = hInfo.biHeight;
  int ImgSize = W*H; // Info헤더에 이미지의 가로/세로 사이즈 정보가 존재한다.
  BYTE* Image = (BYTE*)malloc(ImgSize); // 영상의 크기만큼 동적할당,
  BYTE* Temp = (BYTE*)malloc(ImgSize);
  BYTE* Output = (BYTE*)malloc(ImgSize);
  fread(Image, sizeof(BYTE), ImgSize, fp);// Image 동적할당에 1바이트씩 ImgSize 만큼 fp포인터가 가리키는 곳을 담는다.
  fclose(fp);

  int Histo[256] = { 0 };
  int AHisto[256] = { 0 };
  int NormSum[256];

  /* 2. 영상처리 */

  //InverseImage(Image, Output, W, H); // 1. 영상 반전
  //BrightnessAdj(Image, Output, W,H,70); // 2. 밝기 조절
  //ContrastAdj(Image, Output, W,H,1.5); // 3. 대비 조절

  //ObtainHistogram(Image, Histo, W, H); // 히스토그램 구하기
  //ObtainAHistogram(Histo, AHisto); // 누적 히스토그램 구하기
  //HistogramStretching(Image, Histo, Output, W, H); // 히스토그램 스트레칭
  //HistogramEqualization(Image,Output, AHisto, NormSum, W, H); // 히스토그램 평활화

  //int T = GonzalezBinThresh(Histo); // 임계치 구하기 by. 곤잘레스, 우드 방법
  //Binarization(Image, Output, W, H, T); //영상 이진화

  //AverageConv(Image, Output, W, H);
  //GaussAvrConv(Image, Output, W, H);

  // X와 Y의 경계값을 모두 출력하고 싶다. => Temp 동적할당을 활용
  //Sobel_X_Conv(Image, Temp, W, H);
  //Sobel_Y_Conv(Image, Output, W, H);
  //for (int i = 0; i < ImgSize; i++) { // Prewitt-X마스크에서 출력한 경계가 더욱 강하다면, X-마스크 결과를 넣는다.
  //  if (Temp[i] > Output[i]) Output[i] = Temp[i];
  //}

  //Laplace_Conv_DC(Image, Output, W, H);
  //Binarization(Output, Output, W, H,40); //컨볼루션 이후 => 영상 이진화 : 경계를 더욱 또렷하게 검출 가능

  /* Median Filtering
  BYTE temp[9];
  int i, j;
  for (i = 1; i < H - 1; i++) {
    for (j = 1; j < W - 1; j++) {
      temp[0] = Image[(i-1)*W+(j-1)];
      temp[1] = Image[(i-1)*W+j];
      temp[2] = Image[(i-1)*W+(j+1)];
      temp[3] = Image[i*W+(j-1)];
      temp[4] = Image[i*W+j];
      temp[5] = Image[i*W+(j+1)];
      temp[6] = Image[(i+1)*W+(j-1)];
      temp[7] = Image[(i+1)*W+j];
      temp[8] = Image[(i+1)*W+(j+1)];
      Output[i * W + j] = Median(temp, 9);
    }
  }*/

  //AverageConv(Image, Output, W, H); // 가우시안 잡음 -> 평균필터 통해서 잡음 제거


  // 라벨링 과정 1. 이진화 2. 연결성 분석(라벨링) 3. 특징추출
  //Binarization(Image, Temp, W, H, 100);
  //m_BlobColoring(Temp, H, W); // 이진화된 결과를 라벨링에 Input
  //for (int i = 0; i < ImgSize; i++) {
  //  Output[i] = Image[i];
  //}
```

```
  //BinaryImageEdgeDetection(Temp, Output, W, H);

    //HorizontalFlip(Image, W, H);
  //Translation(Image, Output, W, H, 50, 30);


  // 스케일링
  //Scaling(Image, Output, W, H, 0.7,0.7);

  Rotation(Image, Output, W, H, 45);

  SaveBMPFile(hf, hInfo, hRGB, Output, W, H, "output_translate.bmp");
  free(Image); // 메모리 누수 방지
  free(Output);
  free(Temp);
  return 0;
}
```