1. 题目：图像压缩

1.1.功能实现

选择A组方法3，B组方法2。

使用jpeg算法实现图像压缩，并编码到jpg格式，能用Windows 默认的图片查看器成功打开文件。

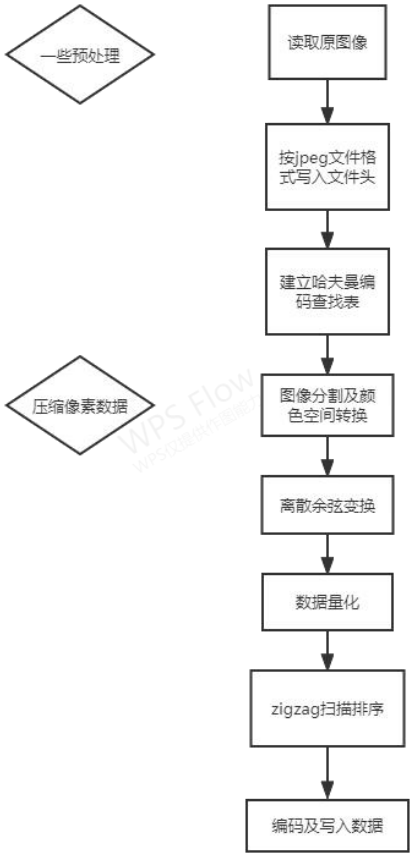
1.2. 注意事项

要求使用面向对象的思想，使用类来封装。

2. 整体设计思路

实现两个功能：采取命令行方式，先在-compress命令下读取原图（lena.tiff），压缩成jpg文件，再在-read命令下读取压缩后的jpg文件并显示在屏幕上。

-compress命令的实现设计思路如下：



3. 主要功能的实现

3.1. jpeg文件头的写入

按照jpeg文件的规定格式写入文件头信息。该过程比较机械，按部就班，但不能出丝毫差错，否则就会导致文件无法打开。

主要包括以下几个部分：SOI，Start of Image，图像开始
APP0，Application，应用程序保留标记0
DQT，Define Quantization Table，定义量化表
SOF0，Start of Frame，帧图像开始
DHT，Difine Huffman Table，定义哈夫曼表
SOS，Start of Scan，扫描开始

3.2. 建立哈夫曼编码查找表

利用四组标准哈夫曼表（亮度直流，色度直流，亮度交流，色度交流），在压缩图像数据之前预处理，建立哈夫曼编码的查找表，压缩图像数据时查表即可得到编码。

具体实现如下：（以亮度直流表为例）

```
const char Standard_DC_Luminance_NRCodes[] = { 0, 0, 7, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 };
    const unsigned char Standard_DC_Luminance_Values[] = { 4, 5, 3, 2, 6, 1, 0, 7, 8, 9, 10, 11 };
```

第一个数组：第i个字符（i从1开始）的值 表示长度为i的编码的个数

第二个数组：value按它的编码长度升序排列

还原编码规则：1.编码从0开始
2.相同长度的编码连续
3.相邻长度的编码的递推规则为上一长度最后一个编码乘2加1

按编码规则从小到大递推，用map存储查找表，下标为待编码的字符，内容为对应的哈夫曼编码。

3.3. 图像分割及颜色空间转换

将图像分成8*8小块，以8*8小块为单位处理数据。

扫描顺序为先第一小块，再第二小块.....每次大循环中，处理一个小块（包括数据处理到写入文件全过程，即边读边写）。

读取到的图像信息是RGBA模式，利用颜色空间转换公式转化为Y,Cb,Cr三个参数（注意：参数Y需要减128，以保证色彩正常）。

3.4. 离散余弦变换及数据量化

$$F(u,v) = alpha(u) * alpha(v) * \sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\left(\frac{2x+1}{16}u\pi\right)\cos\left(\frac{2y+1}{16}v\pi\right) \qquad u,v = 0,1,2,\cdots,7$$

$$alpha(u) = \begin{cases} 1/\sqrt{8} & when\ u = 0 \\ 1/2 & when\ u \neq 0 \end{cases}$$

按离散余弦变换公式计算，再利用量化表（注意对应关系）按公式量化即可。

3.5.zigzag扫描排序

我的方法非常蠢，设置方向参数dx,dy模拟蛇形绕行的过程进行赋值。

更优的方式：全部用一维数组存储，事先打表zigzag数组（第i个元素的值表示蛇形排序后的数组的下标），因此按顺序赋值即可，非常巧妙。

3.6.编码及写入数据

3.6.1. 计算当前8*8小块某分量数组对应的编码

1.由于末尾均为0，首先找到EOB开始的位置并记录。

2.DC部分单独处理：单独处理第一个数据（使用DC部分的哈夫曼表）得到编码。

3.AC部分的编码

RLE编码：将EOB前的数据分块编码，每个单元以非0数结尾或达到单元元素个数上限16，得RLE编码(numof0,val)，表示当前单元最后一个数据为val且前面有numof0个0。

BIT编码：将(numof0,val)中val转为二进制codeval（手动实现，包括负数和0的处理，可找规律）,len为codeval字符串的长度，得(numof0, len, codeval)；由于numof0和len范围为0-15，可合并为一个字节codefirst。

Huffman编码：查找预处理得到的查找表（注意用参数区分亮度色度表），得到codefirst对应的哈夫曼编码，与codeval合并。

末尾加上EOB的编码即为最终编码。

3.6.2. 写入编码

写入顺序：按8*8小块顺序写入，写入每个小块时顺次写入Y,Cb,Cr的编码，每段剩余不足 8 位的编码均必须连入下一段中。

实现：使用一个全局变量字符串tmp存储待写入的编码，tmp中满8位则转为字符输出并清空tmp。（注：遇到ff需要在后面写入00）

3.6.3. 结束

写入EOI扫描结束的标识符ffd9，释放空间等操作。

4. 调试过程碰到的问题（一些坑的总结）

4.1.无法用图片查看器打开lena.jpg文件

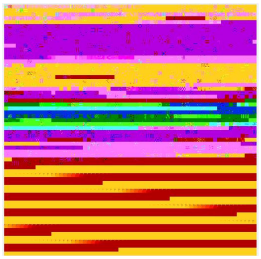
主要原因：文件头写入有错误。

1. 如何写入：使用fout.put()函数全部以单字节方式写入（亲测有效）；另一种方式是采取多字节写入，但要考虑倒序的问题。
2. 文件头中的一些固定值和参数必须保证完全正确，开始时有错误，经反复核对后改正。

4.2.显示的jpg文件错误

1.文件头中写入量化表前需要zigzag扫描排序，否则会导致图像模糊。

2.图像信息数据中，遇到ff要在后面加00。



3.dc部分必须单独处理：一开始的处理在8*8小块的第一个数据为0的情况下会出现与后面ac部分混在一起的情况，必须将dc部分单独处理并注意0的编码等特殊情况是否正确。

4.写入信息的顺序：按8*8小块顺序写入，写入每个小块时顺次写入Y,Cb,Cr的编码，注意各小块之间必须连续。

5.一开始未注意到EOB在亮度色度表中对应的哈夫曼编码不同（不都是1010）。

6.每个8*8小块的第一个数据写入的是偏移量，而不是原值，注意偏移量是当前小块第一个数据减去上一小块第一个数据的原值。

7.颜色空间转换时需要将Y分量减128，否则颜色会有偏差。

4.3.无法用命令行方式read压缩后的jpg文件

问题：可以用图片查看器打开我压缩产生的jpg文件，却不能用命令行方式打开。多次debug无果后，尝试发现我的程序可以通过命令行打开其他jpg文件，别人的程序可以打开我的jpg文件。
解决：最终发现是忘记关闭文件。

5. 心得体会

1. 本次大作业涉及的图像压缩对我来说是一个比较陌生的领域，步骤又比较多，很考验我的自学能力。在这种情况下，要认真学习各种资料，先理清算法原理和整体思路。

2. 本次大作业要求使用面向对象思想编写，让我进一步实践了类和对象的使用，也体会到了用类和对象封装的优越性。

3. 这次大作业debug过程异常艰辛，写bug一天半，debug两天半。主要是这次算法比较复杂，步骤多，而且有很多细节问题。小问题不容易发现，还有一些坑很容易踩到甚至往往不经提醒很难发现。很考验耐心和毅力。同时，也教会我一步步分块debug，提高了debug复杂程序的能力。

附件：源程序

/*jpeg.cpp*/

```
 #include "PicReader.h"
#include "huffmancode.h"
#include<iostream>
#include <fstream>
#include <stdio.h>
#include<cmath>
#include<map>
#include<bitset>
using namespace std;

class jpegcompress {
    private:
        /*BYTE*&, 传入 BYTE 指针引用,用于接收像素信息
        * img_width  图像的宽度信息
        * img_height  图像的高度信息
          *  像素信息每四个一组(R G B A)       */
        PicReader imread;
        BYTE* data = nullptr;
        UINT img_width, img_height;

        //哈夫曼编码查找表
        map<unsigned char, string> mp_DC_lumin;    //亮度直流
        map<unsigned char, string> mp_AC_lumin;    //亮度交流
        map<unsigned char, string> mp_DC_chro;     //色度直流
        map<unsigned char, string> mp_AC_chro;     //色度交流

    public:
        string tmp = "";    //存放写入数据时的 8 位 01 串

        //读入图片
        void read_picture(char* argv[]);
        //文件头的写入
        void prewrite(ofstream &fout);
        //建立四张 huffman 编码查找表
        void huffmancode();
        //建立 huffman 编码查找表
        void set_huffmancode_list(int chooselist, const char*
code_cnt, const unsigned char* code_value);
        //写入压缩后的图像信息
        void writecode(int zz[64], int chooselist, ofstream &fout);
        //得到该 8*8 小块的 01 串编码
        void getcode(int zz[64], int chooselist, string& result);
        //处理原图像素信息
        void process_pixel(ofstream &fout);
        //写入文件结束及释放空间
        void end(ofstream &fout);
};

int zz_Y[N * N] = { 0 }, zz_Cb[N * N] = { 0 }, zz_Cr[N * N] = { 0 };
//zigzag 扫描排序后的数组
int preY = 0, preCb = 0, preCr = 0;    //存放上一小块第一个数据的
原值（用于差分）

//读入图片
void jpegcompress::read_picture(char* argv[])
{
    imread.readPic(argv[2]);
    imread.getData(data, img_width, img_height);
}

//计算每小段 RLE 编码的第二个数对应的编码
string getcode_second(int val)
{
    string codeval = "";
    for (int k = abs(val); k > 0; k = k / 2) {
        codeval = codeval + (k % 2 == 1 ? '1' : '0');
    }
    if (val < 0)   //处理负数的情况
        for (int k = 0; k < (int)codeval.length(); k++)
            if (codeval[k] == '1')
                codeval[k] = '0';
            else
                codeval[k] = '1';

    //字符串倒序
    for (int i = 0; i < (int)codeval.length() / 2; i++) {
        char ch = codeval[i];
        codeval[i] = codeval[codeval.length() - i - 1];
        codeval[codeval.length() - i - 1] = ch;
    }
    return codeval;
}

static string to_binary_str(int code, int bitslen, string buf) {
    int mask = 1 << (bitslen - 1);
    for (int i = 0; i < bitslen; i++) {
        if (code & mask)
            buf += '1';
        else
            buf += '0';
        mask >>= 1;
    }
    buf[bitslen] = '0';
    return buf;
}

//建立 huffman 编码查找表
//chooselist 区分直流交流亮度色度，code_cnt 指向不同长度编码
的个数数组，code_value 为待编码的字符
void jpegcompress::set_huffmancode_list(int chooselist, const char*
code_cnt, const unsigned char* code_value)
{
    int sym = 0, code = 0;
    string buf = "";
    for (int i = 0; i < 16; i++) {
        int num = (int)code_cnt[i];
        int bitslen = i + 1;
        for (int j = 0; j < num; j++) {
            unsigned char symbol = code_value[sym];
            string result = to_binary_str(code, bitslen, buf);

            if (chooselist == 1)
                mp_DC_lumin.insert(make_pair(symbol,
result));
            else if (chooselist == 2)
                mp_AC_lumin.insert(make_pair(symbol,
result));
            else if (chooselist == 3)
                mp_DC_chro.insert(make_pair(symbol, result));
            else
                mp_AC_chro.insert(make_pair(symbol, result));

            code++;
            sym++;
        }
        code <<= 1;
    }
}

//建立四张 huffman 编码查找表
void jpegcompress::huffmancode()
{
    set_huffmancode_list(1, DC_lumin, DC_lumin_value);
    set_huffmancode_list(2, AC_lumin, AC_lumin_value);
    set_huffmancode_list(3, DC_chro, DC_chro_value);
    set_huffmancode_list(4, AC_chro, AC_chro_value);
}

//得到该 8*8 小块的 01 串编码
//zz 为以 zigzag 方式排列成的一维数组，chooselist 区分直流交流
亮度色度，result 储存编码
void jpegcompress::getcode(int zz[64], int chooselist, string& result)
```

```
{
    result = "";
    //numof0 表示该小段最后一个数前面 0 的个数，val 表示该段末尾的数
    int numof0 = 0, val = 0;
    int end = 64;   //end 表示 EOB 的位置
    for (int i = 63; i >= 0; i--)
        if (zz[i] != 0) {
            end = i + 1;
            break;
        }

    //DC 部分
    string codeval = getcode_second(zz[0]);
    int len = codeval.length();
    if (len != 0) {
        if (chooselist == 1)
            result += mp_DC_lumin[(unsigned char)len];
        else
            result += mp_DC_chro[(unsigned char)len];
    }
    else {
        if (chooselist == 1)
            result += mp_DC_lumin[(unsigned char)0x00];
        else
            result += mp_DC_chro[(unsigned char)0x00];
    }
    result += codeval;

    //AC 部分
    for (int i = 1; i < end; i++) {
        if (zz[i] == 0 && numof0 < 15)   //每小段以非 0 数结尾或者最多 16 个数
            numof0++;
        else {
            val = zz[i];
            string codeval = getcode_second(val);   //每小段 RLE 编码的第二个数对应的编码
            int len = codeval.length();   //BIT 编码的中间一个数


            if (chooselist == 1) {
                result += mp_AC_lumin[(unsigned char)((numof0 << 4) + len)];
            }
            else {
                result += mp_AC_chro[(unsigned char)((numof0 << 4) + len)];
            }
            result += codeval;

            numof0 = 0;
        }
    }
}

//写入哈夫曼编码表
void print_huffman(char tag, const char   valuelen[], const unsigned char value[], ofstream &fout)
{
    fout.put(tag);   //表 ID 和表类型
    int sum = 0;      //数组 value 的元素个数为 valuelen 所有元素之和
    for (int i = 0; i < 16; i++) {
        fout.put(valuelen[i]);
        sum += (int)valuelen[i];
    }
    for (int i = 0; i < sum; i++)
        fout.put(value[i]);
}

//以蛇形输出量化表
void print_Q(const unsigned char Q[N][N], ofstream &fout)
{
    int tx = 0, ty = 0, dx = -1, dy = 1, cnt = 0;
    while (tx != N - 1 || ty != N - 1) {
        fout.put(Q[tx][ty]);
        if ((tx == 0 || tx == N - 1) && ty % 2 == 0) {
            ty++;
            dx *= -1;
            dy *= -1;
            continue;
        }
        if ((ty == 0 || ty == N - 1) && tx % 2 == 1) {
            tx++;
            dx *= -1;
            dy *= -1;
            continue;
        }
        tx += dx;
        ty += dy;
    }
    fout.put(Q[N - 1][N - 1]);
}

//文件头的写入
void jpegcompress::prewrite(ofstream &fout)
{
    /*SOI 图像开始*/
    fout.put((char)0xff), fout.put((char)0xd8);
    fout.put((char)0xff), fout.put((char)0xe0);   //APP0
    fout.put(0x00), fout.put(0x10);
    fout.put(0x4a), fout.put(0x46), fout.put(0x49), fout.put(0x46), fout.put(0x00);
    fout.put(0x01), fout.put(0x01), fout.put(0x00), fout.put(0x00), fout.put(0x01), fout.put(0x00), fout.put(0x01);
    fout.put(0x00), fout.put(0x00);

    /*DQT 量化表*/
    fout.put((char)0xff), fout.put((char)0xdb);   //标记代码
    fout.put((char)0x00), fout.put((char)0x84);   //数据长度(包括本字段)
    fout.put(0x00);   //精度及量化表 ID
    print_Q(Qy, fout);      //以蛇形输出量化表 Qy
    fout.put(0x01);   //精度及量化表 ID
    print_Q(Qc, fout);      //以蛇形输出量化表 Qc

    /*SOF0 帧图像开始*/
    fout.put((char)0xff), fout.put((char)0xc0);   //标记代码
    fout.put(0x00), fout.put(0x11), fout.put(0x08);
    fout.put(0x02); fout.put(0x00); //fout << img_height;   //图像高度(2 字节)
    fout.put(0x02); fout.put(0x00); //fout << img_width;   //图像宽度(2 字节)
    fout.put(0x03);   //颜色分量数
    //颜色分量信息
    fout.put(0x01), fout.put(0x11), fout.put(0x00);
    fout.put(0x02), fout.put(0x11), fout.put(0x01);
    fout.put(0x03), fout.put(0x11), fout.put(0x01);

    /*DHT 定义哈夫曼表*/
    fout.put((char)0xff), fout.put((char)0xc4);   //标记代码
    fout.put((char)0x01), fout.put((char)0xa2);   //数据长度
    print_huffman(0x00, DC_lumin, DC_lumin_value, fout);
    print_huffman(0x10, AC_lumin, AC_lumin_value, fout);
    print_huffman(0x01, DC_chro, DC_chro_value, fout);
    print_huffman(0x11, AC_chro, AC_chro_value, fout);

    /*SOS 扫描开始*/
    fout.put((char)0xff), fout.put((char)0xda);   //标记代码
    fout.put(0x00), fout.put(0x0c);   //数据长度
    fout.put(0x03);   //颜色分量数
    fout.put(0x01), fout.put(0x00), fout.put(0x02), fout.put(0x11), fout.put(0x03), fout.put(0x11);   //颜色分量信息 (?)
    fout.put(0x00), fout.put(0x3f), fout.put(0x00);   //固定值
}

//写入压缩后的图像信息
```

```
//zz 为以 zigzag 方式排列成的一维数组，chooselist 区分直流交流亮度色度
void jpegcompress::writecode(int zz[64], int chooselist, ofstream &fout)
{
    string result = "";
    getcode(zz, chooselist, result);   //获取 EOB 前的编码，存入字符串 result 中
    if (chooselist == 1)
        result += mp_AC_lumin[(unsigned char)0x00];
    else
        result += mp_AC_chro[(unsigned char)0x00];
    //cout << result;
    //将 result 按单字节写入文件
    for (int pos = 0; pos < (int)result.length(); pos++) {
        tmp += result[pos];
        if ((int)tmp.length() >= 8) {    //tmp 满 8 位则将该字节编码转为字符输出
            //cout << n << endl;
            bitset<8> n(tmp);
            unsigned char ch = (unsigned char)(n.to_ulong());
            fout.put(ch);
            if (ch == 0xff)
                fout.put(0x00);
            tmp = "";    //清空 tmp
        }
    }
}

double alpha(int x)
{
    if (x == 0)
        return 1.0 / sqrt(8);
    else
        return 0.5;
}

//处理原图像素信息
void jpegcompress::process_pixel(ofstream &fout)
{
    int count = 0;   //当前第 count 个 8*8 小块
    for (DWORD i = 0; i < img_height; i += 8)
        for (DWORD j = 0; j < img_width; j += 8) {
            //图像分割：对每个 8*8 小块
            count++;
            double Y[N][N] = { 0 }, Cb[N][N] = { 0 }, Cr[N][N] = { 0 };
            for (DWORD u = 0; u < 8; u++)
                for (DWORD v = 0; v < 8; v++) {
                    int k = (i + u) * img_width * 4 + (j + v) * 4;
                    //颜色空间转换 RGB->YCbCr
                    int R = data[k], G = data[k + 1], B = data[k + 2], A = data[k + 3];
                    Y[u][v] = 0.29871 * R + 0.58661 * G + 0.11448 * B - 128;
                    Cb[u][v] = -0.16874 * R - 0.33126 * G + 0.5 * B;
                    Cr[u][v] = 0.5 * R - 0.41869 * G - 0.08131 * B;
                }

            //离散余弦变换
            double f_Y[N][N] = { 0 }, f_Cb[N][N] = { 0 }, f_Cr[N][N] = { 0 };
            for (int u = 0; u < N; u++)
                for (int v = 0; v < N; v++) {
                    double tmp1 = 0, tmp2 = 0, tmp3 = 0;
                    for (int tu = 0; tu < N; tu++)
                        for (int tv = 0; tv < N; tv++) {
                            tmp1 += Y[tu][tv] * cos(1.0*(2 * tu + 1) * u * Pi / 16.0) * cos(1.0*(2 * tv + 1) * v * Pi / 16.0);
                            tmp2 += Cb[tu][tv] * cos(1.0*(2 * tu + 1) * u * Pi / 16.0) * cos(1.0*(2 * tv + 1) * v * Pi / 16.0);
                            tmp3 += Cr[tu][tv] * cos(1.0*(2 * tu + 1) * u * Pi / 16.0) * cos(1.0*(2 * tv + 1) * v * Pi / 16.0);
                        }
                    f_Y[u][v] = alpha(u) * alpha(v) * tmp1;
                    f_Cb[u][v] = alpha(u) * alpha(v) * tmp2;
                    f_Cr[u][v] = alpha(u) * alpha(v) * tmp3;
                }

            //数据量化
            int qua_Y[N][N], qua_Cb[N][N], qua_Cr[N][N];
            for (int u = 0; u < N; u++)
                for (int v = 0; v < N; v++) {
                    qua_Y[u][v] = (int)round(f_Y[u][v] / (int)Qy[u][v]);
                    qua_Cb[u][v] = (int)round(f_Cb[u][v] / (int)Qc[u][v]);
                    qua_Cr[u][v] = (int)round(f_Cr[u][v] / (int)Qc[u][v]);
                }

            //Zigzag 扫描排序 (把量化后的二维矩阵转变成一个一维数组)

            int tmpY = qua_Y[0][0];
            int tmpCb = qua_Cb[0][0];
            int tmpCr = qua_Cr[0][0];
            zz_Y[0] = qua_Y[0][0] - preY;
            zz_Cb[0] = qua_Cb[0][0] - preCb;
            zz_Cr[0] = qua_Cr[0][0] - preCr;
            preY = tmpY;
            preCb = tmpCb;
            preCr = tmpCr;

            int tx = 0, ty = 1, dx = 1, dy = -1, cnt = 1;
            while (tx != N - 1 || ty != N - 1) {
                zz_Y[cnt] = qua_Y[tx][ty];
                zz_Cb[cnt] = qua_Cb[tx][ty];
                zz_Cr[cnt] = qua_Cr[tx][ty];
                cnt++;
                if ((tx == 0 || tx == N - 1) && ty % 2 == 0) {
                    ty++;
                    dx *= -1;
                    dy *= -1;
                    continue;
                }
                if ((ty == 0 || ty == N - 1) && tx % 2 == 1) {
                    tx++;
                    dx *= -1;
                    dy *= -1;
                    continue;
                }
                tx += dx;
                ty += dy;
            }
            zz_Y[cnt] = qua_Y[N - 1][N - 1];
            zz_Cb[cnt] = qua_Cb[N - 1][N - 1];
            zz_Cr[cnt] = qua_Cr[N - 1][N - 1];

            //写入压缩后的图像信息
            writecode(zz_Y, 1, fout);
            writecode(zz_Cb, 2, fout);
            writecode(zz_Cr, 2, fout);
        }
        if (tmp != "") {    //输出最后剩余不足 8 位的
            bitset<8> n(tmp);
            fout.put((unsigned char)n.to_ulong());
        }
}

//写入文件结束及释放空间
void jpegcompress::end(ofstream &fout)
{
    fout.put((char)0xff);
    fout.put((char)0xd9);
    delete[] data;
    data = nullptr;
}
```

```
int main(int argc, char* argv[])
{
    if (argc != 3) {
        cerr << "Please make sure the number of parameters is correct." << endl;
        return -1;
    }
    if (strcmp(argv[1], "-read") && strcmp(argv[1], "-compress")) {
        cerr << "Unknown parameter!\nCommand list:\nzip/unzip" << endl;
        return -1;
    }

    jpegcompress jpeg;

    if (strcmp(argv[1], "-compress") == 0) {
        //文件打开
        ifstream fin(argv[2], ios::binary); //  以二进制方式打开输入文件
        if (!fin) {    // 输出错误信息并退出
            cerr << "Can not open the input file!" << endl;
            return -1;
        }

        ofstream fout("lena.jpg", ios::binary);
        if (!fout) {
            cerr << "Can not open the output file!" << endl;
            return -1;
        }

        jpeg.read_picture(argv);
        jpeg.prewrite(fout);
        jpeg.huffmancode();
        jpeg.process_pixel(fout);
        jpeg.end(fout);
        cout << "complete!" << endl;

        fin.close();
        fout.close();
    }
    else if (strcmp(argv[1], "-read") == 0) {
        PicReader imread;
        BYTE* data = nullptr;
        UINT x, y;

        imread.readPic(argv[2]);
        imread.getData(data, x, y);
        imread.showPic(data, x, y);
        delete[] data;
    }

    return 0;
}
```

```
/*huffmancode.h*/
#pragma once
#include<string>
using namespace std;

#define N 8
const double Pi = acos(-1);

const unsigned char Qy[N][N] = { 16,11,10,16,24,40,51,61,

12,12,14,19,26,58,60,55,

14,13,16,24,40,57,69,56,

14,17,22,29,51,87,80,62,

18,22,37,56,68,109,103,77,

24,35,55,64,81,104,113,92,

49,64,78,87,103,121,120,101,

72,92,95,98,112,100,103,99 };

const unsigned char Qc[N][N] = { 17,18,24,47,99,99,99,99,

18,21,26,66,99,99,99,99,

24,26,56,99,99,99,99,99,

47,66,99,99,99,99,99,99,

99,99,99,99,99,99,99,99,

99,99,99,99,99,99,99,99,

99,99,99,99,99,99,99,99,

99,99,99,99,99,99,99,99 };

const char DC_lumin[] = { 0, 0, 7, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 };
const unsigned char DC_lumin_value[] = { 4, 5, 3, 2, 6, 1, 0, 7, 8, 9, 10, 11 };

const char DC_chro[] = { 0, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0 };
const unsigned char DC_chro_value[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };


//第 i 个字符（i 从 1 开始）的值 表示长度为 i 的编码的个数
const char AC_lumin[] = { 0, 2, 1, 3, 3, 2, 4, 3, 5, 5, 4, 4, 0, 0, 1, 0x7d };
//value 按它的编码长度升序排列
const unsigned char AC_lumin_value[] =
{
        0x01, 0x02, 0x03, 0x00, 0x04, 0x11, 0x05, 0x12,
        0x21, 0x31, 0x41, 0x06, 0x13, 0x51, 0x61, 0x07,
        0x22, 0x71, 0x14, 0x32, 0x81, 0x91, 0xa1, 0x08,
        0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52, 0xd1, 0xf0,
        0x24, 0x33, 0x62, 0x72, 0x82, 0x09, 0x0a, 0x16,
        0x17, 0x18, 0x19, 0x1a, 0x25, 0x26, 0x27, 0x28,
        0x29, 0x2a, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,
        0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49,
        0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,
        0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
        0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
        0x7a, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
        0x8a, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98,
        0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
        0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6,
        0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3, 0xc4, 0xc5,
        0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2, 0xd3, 0xd4,
        0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xe1, 0xe2,
        0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea,
        0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
        0xf9, 0xfa
};


const char AC_chro[] = { 0, 2, 1, 2, 4, 4, 3, 4, 7, 5, 4, 4, 0, 1, 2, 0x77 };
const unsigned char AC_chro_value[] = { 0x00, 0x01, 0x02, 0x03, 0x11, 0x04, 0x05, 0x21,
        0x31, 0x06, 0x12, 0x41, 0x51, 0x07, 0x61, 0x71,
        0x13, 0x22, 0x32, 0x81, 0x08, 0x14, 0x42, 0x91,
        0xa1, 0xb1, 0xc1, 0x09, 0x23, 0x33, 0x52, 0xf0,
        0x15, 0x62, 0x72, 0xd1, 0x0a, 0x16, 0x24, 0x34,
        0xe1, 0x25, 0xf1, 0x17, 0x18, 0x19, 0x1a, 0x26,
        0x27, 0x28, 0x29, 0x2a, 0x35, 0x36, 0x37, 0x38,
        0x39, 0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48,
        0x49, 0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
        0x59, 0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68,
        0x69, 0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78,
        0x79, 0x7a, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
        0x88, 0x89, 0x8a, 0x92, 0x93, 0x94, 0x95, 0x96,
        0x97, 0x98, 0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5,
        0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4,
```

```
        0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3,
        0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2,
        0xd3, 0xd4, 0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda,
        0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9,
        0xea, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
        0xf9, 0xfa
};
```