同济大学计算机系

OOP 图像压缩大作业实验报告

学　　号 2152118

姓　　名 史君宝

专　　业 计算机科学与技术（计科 1 班）

完成时间 2023.12.15

一、设计思路与功能描述

1. 得分点

实现了将 lena.tiff 文件转换为 lena.jpg 文件
完成了 A 任务的 50%和 B 任务的 40%。

2. 设计思路

（1）采用标准的 jpg 图像压缩过程进行图像压缩的编码

在图像压缩中有很多的压缩格式，我们今天主要使用的是 jpg 型压缩的压缩格式。其中主要过程有 DCT 转化、数据量化和 Zigzag 排序还有经典的 Huffman 编码过程。

我们将具体讲述整个 jpg 图像压缩的具体过程，然后实现将 lena.tiff 转换为 lena.jpg。

3. 功能描述

（1）全局变量的声明：

由于我们在网上查找的资料可知，jpg 的压缩有一个具体明确的过程，我们仅仅是复现这一过程。中间会用到经常使用的一个数组。

```
//PEG算法提供了两张标准化系数矩阵，分别处理亮度数据和色差数据
//表1
const int Qy[8][8] = {
    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
};
```

```
//表2
const int Qc[8][8] = {
    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
};
```

```
//ZIGZAG排序数组
const char ZIGZAG[64] = {
    0,  1,  5,  6,  14,  15,  27,  28,
    2,  4,  7,  13,  16,  26,  29,  42,
    3,  8,  12,  17,  25,  30,  41,  43,
    9,  11,  18,  24,  31,  40,  44,  53,
    10,  19,  23,  32,  39,  45,  52,  54,
    20,  22,  33,  38,  46,  51,  55,  60,
    21,  34,  37,  47,  50,  56,  59,  61,
    35,  36,  48,  49,  57,  58,  62,  63
};
```

```
//DC，AC编码数组
BYTE Standard_DC_Luminance_NRCodes[] = { 0,  0,  7,  1,  1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0 };
BYTE Standard_DC_Luminance_Values[] = { 4,  5,  3,  2,  6,  1,  0,  7,  8,  9,  10,  11 };
BYTE Standard_DC_Chrominance_NRCodes[] = { 0,  3,  1,  1,  1,  1,  1,  1,  1,  1,  1,  0,  0,  0,  0,  0 };
BYTE Standard_DC_Chrominance_Values[] = { 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10,  11 };
BYTE Standard_AC_Luminance_NRCodes[] = { 0,  2,  1,  3,  3,  2,  4,  3,  5,  5,  4,  4,  0,  0,  1,  0x7d };
BYTE Standard_AC_Luminance_Values[] = { ... }
BYTE Standard_AC_Chrominance_NRCodes[] = { 0,  2,  1,  2,  4,  4,  3,  4,  7,  5,  4,  4,  0,  1,  2,  0x77 };
BYTE Standard_AC_Chrominance_Values[] = { ... }
```

（2）main 函数内容：

功能:实现一个简单的程序选择结构，根据命令行的输入指令，转向对应的压缩或者读取过程：

```
//主函数
int main(int argc, char** argv) {
    cout << "开始运行" << endl;

    if (argc != 3) {
        cerr << "请检查参数个数是否正确" << endl;
        return -1;
    }

    if (!strcmp(argv[1], "-compress") || !strcmp(argv[1], "-read")) {
        if (!strcmp(argv[1], "-compress"))
        {
            Compress_pic(argv[2]);
        }
        else if (!strcmp(argv[1], "-read"))
        {
            Read_pic(argv[2]);
        }
    }
    else {
        cerr << "Unknown parameter!\nCommand list:\n-compress\n-read" << endl;
        return -1;
    }

    cout << "Complete!" << endl;
    return 0;
}
```

（3）函数声明：

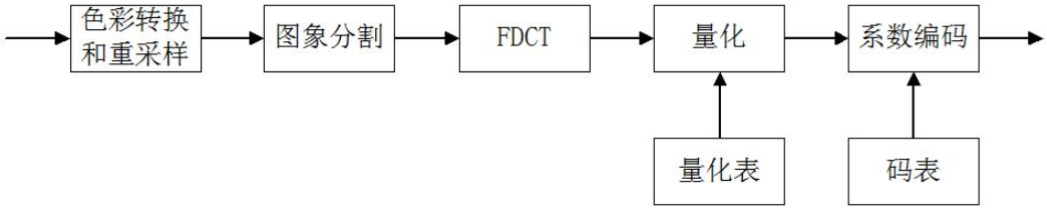
```
//压缩图片函数
void Compress_pic(const char* filename);
//读图片函数
void Read_pic(const char* filename);
//将压缩过程中读到的数据data分块为8*8
int div_88(compress_block* head, BYTE*& data, int x, int y);
//为每一块进行计算，算出YCbCr
void RGB_cpu(compress_block* new_node);
//获得压缩结果字符串result的函数
void Get_result(int img_height, int img_width, compress_block* head);
//具体的压缩编码过程
void compress_process(compress_block* head);
//压缩编码结束后，需要将Huffman编码按照8位转换为新字符
void write_byte();
```

```
//下面详述压缩过程
//DCT过程
void DCT(double f[8][8], double F[8][8]);
//DCT中用到alpha函数
double alpha(int u);
//数据量化过程
void quantify(double YCbCr[8][8], const int Q[8][8], int* tmp);
//Zigzag排序过程
void Zigzag_sort(int tmp[64]);
//获得具体编码过程
void writeCode(int* tmp, int DC_AC_type, int num);

//初始化Huffman编码
void Init_Huffman_Code();
//根据DC，AC系数表获得Huffman编码
void Get_Huffman_Code(BYTE* NRCodes, BYTE* Values, int index);

int Get_BitLen(int bit);
//动态内存申请之后的释放函数
void delete_block(compress_block* head);
```

（4）外部资料查找的压缩过程：

我们先从外部查找的资料具体讲述一下压缩的全过程，之后会结合自己的代码，说说具体的过程都是如何实现的。

基本是下面的一个过程。



首先我们需要进行颜色的采样，即获得 RGB 的颜色，这我们在下面的步骤中就可以轻松的获得：

*

由于读出来的数据为一维数组，故还提供了原始文件的宽度和高度，你可以采用如下方法索引该图片任意位置的像素值。

```
int index = row * width * 4 + col * 4;
BYTE r = data[index];
BYTE g = data[index + 1];
BYTE b = data[index + 2];
BYTE a = data[index + 3];
```

但是 jpg 压缩过程主要是对 8*8 的块进行操作，所以我们需要进行一个图像的分割，即图像分割过程。

然后我们需要将 8*8 块的 RGB 转换为 YCbCr 值，具体过程在网上都有公式：

Y = 0.299R + 0.587G + 0.114B

U = 0.5R - 0.4187G - 0.0813G + 128

V = -0.1687R - 0.3313G + 0.5B + 128

之后我们需要对上面获得的 YCbCr 值进行一个 DCT（离散余弦变换）的过程：

二维离散余弦变换公式为：

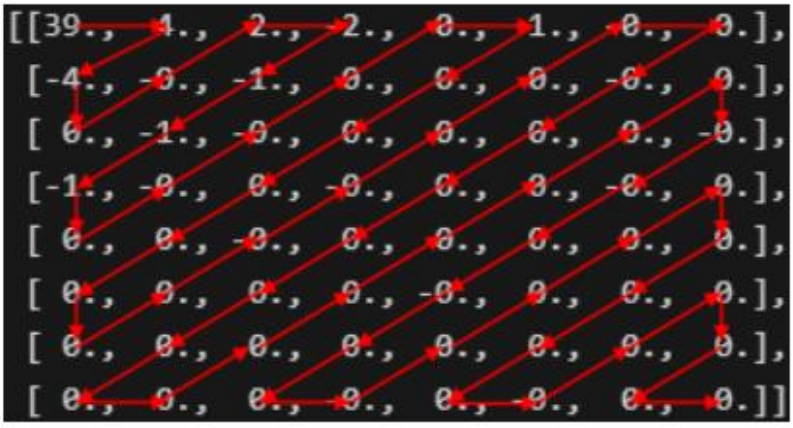
$$F(x,y) = alpha(x) * alpha(y) * \sum_{i=0}^{7}\sum_{j=0}^{7} f(i,j)cos(\frac{2i+1}{16}x\pi)cos(\frac{2j+1}{16}y\pi) \qquad x,y = 0,1,\ldots,7$$

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

DCT 之后我们需要进行数据量化
用到的就是我们声明的全局变量两个数组 Qy 和 Qc

$$Q_Y = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \qquad Q_C = \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

之后我们进行 Zigzag 排序，其排序的要求为：



我们使用到的数组也是 Zigzag 排序数组：

```
//ZIGZAG排序数组
const char ZIGZAG[64] = {
    0,  1,  5,  6,  14,  15,  27,  28,
    2,  4,  7,  13,  16,  26,  29,  42,
    3,  8,  12,  17,  25,  30,  41,  43,
    9,  11,  18,  24,  31,  40,  44,  53,
    10,  19,  23,  32,  39,  45,  52,  54,
    20,  22,  33,  38,  46,  51,  55,  60,
    21,  34,  37,  47,  50,  56,  59,  61,
    35,  36,  48,  49,  57,  58,  62,  63
};
```

之后我们会根据常用的编码方式，获得一个具体的 Huffman 编码表，这里就不再赘述了，我们讲一下编码过程的逻辑。
首先在上面的处理之后我们会获得原始数据：

我们回到JPEG压缩上，回顾上一节的内容，经过数据量化，我们现在要处理的数据是一串一维数组，举例如下：

①原始数据	35,7,0,0,0,-6,-2,0,0,-9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,0,0,0,...,0

之后我们需要将其转换为 RLE 编码：

②RLE编码 35 7 0,0,0,-6 -2 0,0,-9 0,0,...,0,8 0,0,...,0

如果其中某个单元的0的个数超过16，则需要分成每16个一组，如果最后一个单元全都是0，则使用特殊字符"EOB"表示，EOB意思就是"后面的数据全都是0"，

①原始数据	35,7,0,0,0,-6,-2,0,0,-9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,0,0,0,...,0

②RLE编码 35 7 0,0,0,-6 -2 0,0,-9 0,0,...,0,8 0,0,...,0 35 7 0,0,0,-6 -2 0,0,-9 0,0,...,0 0,0,8 0,0,...,0 (0,35) (0,7) (3,-6) (0,-2) (2,-9) (15,0) (2,8) EOB

RLE 编码之后我们还有将其转换为 BIT 编码

②RLE编码 35 7 0,0,0,-6 -2 0,0,-9 0,0,…,0,8 0,0,…,0 35 7 0,0,0,-6 -2 0,0,-9 0,0,…,0 0,0,8 0,0,…,0 (0,35) (0,7) (3,-6) (0,-2) (2,-9) (15,0) (2,8) EOB **③BIT编码** (0,6, ***100011***) (0,3, ***111***) (3,3, ***001***) (0,2, ***01***) (2,4, ***0110***) (15,-) (2,4, ***1000***) EOB

括号中前两个数字分都在0~15之间，所以这两个数可以合并成一个byte，高四位是前面0的个数，后四位是后面数字的位数。

最后使用我们之前获得的 Huffman 编码就会获得 01 序列了。
具体过程可以总结为：

①原始数据	35,7,0,0,0,-6,-2,0,0,-9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,0,0,0,...,0							
②RLE编码	35	7	0,0,0,-6	-2	0,0,-9	0,0,...,0,8		0,0,...,0
	35	7	0,0,0,-6	-2	0,0,-9	0,0,...,0	0,0,8	0,0,...,0
	(0,35)	(0,7)	(3,-6)	(0,-2)	(2,-9)	(15,0)	(2,8)	EOB
③BIT编码	(0,6, ***100011***)	(0,3, ***111***)	(3,3, ***001***)	(0,2, ***01***)	(2,4, ***0110***)	(15,-)	(2,4, ***1000***)	EOB
	(0x6,***100011***)	(0x3,***111***)	(0x33,***001***)	(0x2,***01***)	(0x24,***0110***)	0xF0	(0x24,***1000***)	EOB
④哈弗曼编码	***100*** ***100011***	***100*** ***111***	***1111 1111 0101*** ***001***	***01*** ***01***	***1111 1111 0100*** ***0110***	***1111 1111 001***	***1111 1111 0100*** ***1000***	***1010***
⑤序列化	10010001110011111111111010100101011111111110100011011111111100111111111010010001010							
	91 CF FE A5 7F D1 BF CF FA 45							

之后我们将上面获得的 01 序列每 8 位进行一个分割，转换对应的字符就可以了。

当然对于 jpg 的文件我们还有具体的格式要求，所以我们需要将一些符合格式的字符输入到最终结果中：
具体的格式有：

```
--------------------------------------
名称   标记码      说明
--------------------------------------
SOI      D8           文件头
EOI      D9           文件尾
SOF0     C0           帧开始（标准 JPEG）
SOF1     C1           同上
DHT      C4           定义 Huffman 表（霍夫曼表）
SOS DA             扫描行开始
DQT DB             定义量化表
DRI      DD           定义重新开始间隔
APP0     E0           定义交换格式和图像识别信息
COM FE             注释
----------------------------------------------------------
```

这些东西不太好确定，我们最好把文件转换为jpg之后打开具体看看是怎么样的。之后将这些格式内容加入到最终的结果中就可以了。

（5）结合自身代码，整个压缩过程：

首先是划分为 8*8 的块，我们将数据分为 8*8 的块，然后获得其中的 RGB 值，然后构成一个链表结构。

具体的是实现过程是使用下面的 div_88 的函数，逐个的申请块结构，然后构成链表。

```
//将所有的数据划分为8*8的块的链表
int div_88(compress_block* head, BYTE*& data, int x, int y)
{
    compress_block* curnode;

    //数据转换成8*8的整块
    for (int i = 0; i < x; i += 8)
    {
        for (int j = 0; j < y; j += 8)
        {
            compress_block* new_node = new compress_block();
            if (!new_node)
            {
                cout << "动态内存申请失败" << endl;
                return -1;
            }
```

获得 RGB 的过程：

```
            //将这一8*8块的RGB赋值进结构体中
            for (int p = i; p < i + 8; ++p)
            {
                for (int q = j; q < j + 8; ++q)
                {
                    int index = p * y * 4 + q * 4;
                    new_node->R[p - i][q - j] = data[index];
                    new_node->G[p - i][q - j] = data[index + 1];
                    new_node->B[p - i][q - j] = data[index + 2];

                }
            }
```

将 RGB 的值转换为 YCbCr 值：

```
void RGB_cpu(compress_block* new_node)
{
    for (int i = 0; i < 8; ++i)
    {
        for (int j = 0; j < 8; ++j)
        {
            new_node->Y[i][j] = 0.29871 * new_node->R[i][j] + 0.58661 * new_node->G[i][j] + 0.11448 * new_node->B[i][j]
            new_node->Cb[i][j] = -0.16874 * new_node->R[i][j] - 0.33126 * new_node->G[i][j] + 0.50000 * new_node->B[i][j
            new_node->Cr[i][j] = 0.50000 * new_node->R[i][j] - 0.41869 * new_node->G[i][j] - 0.08131 * new_node->B[i][j]
        }
    }

    return;
}
```

我们使用的也是之前提到的转换公式。

然后我们先将具体的 jpg 格式输入进去，具体为：
SOI 部分：

```
    //SOI文件头
    result += char(0xff);
    result += char(0xd8);
```

APP0 部分：

```
//APP0图像识别信息
//图像识别信息头
result += char(0xff);
result += char(0xe0);
//段长度
result += char(0x00);
result += char(0x10);
//交换格式
result += char(0x4A);
result += char(0x46);
result += char(0x49);
result += char(0x46);
result += char(0x00);
//主版本号和次版本号
result += char(0x01);
result += char(0x02); //不一样
//单位密度
result += char(0x00); //不一样
//X像素密度
result += char(0x00); //不一样
result += char(0x01);
//Y像素密度
result += char(0x00); //不一样
```

DQT 部分：

```
//DQT定义量化表
//定义量化表的头
result += char(0xff);
result += char(0xdb);
//段长度
result += char(0x00);
result += char(0x84);
//QT信息
result += char(0x00);
//QT量化表 量化表0
int Zigzag_1[64], Zigzag_2[64];
for (int i = 0; i < 8; i++)
    for (int j = 0; j < 8; j++)
        Zigzag_1[i * 8 + j] = Qy[i][j];
for (int i = 0; i < 64; i++)
    Zigzag_2[ZIGZAG[i]] = Zigzag_1[i];
for (int i = 0; i < 64; i++)
    result += char(Zigzag_2[i]);
```

S0F0 部分：

```
//SOF0图像基本信息
//图像基本信息
result += char(0xff);
result += char(0xc0);
//段长度
result += char(0x00);
result += char(0x11);
//样本精度
result += char(0x08);
//样本高度
result += char(img_height >> 8);
result += char(img_height & 0b1111);
//样本宽度
result += char(img_width >> 8);
result += char(img_width & 0b1111);
//组件数量
result += char(0x03);
//Y组件
result += char(0x01); //组件ID
```

EOI 文件尾：

```
//EOI文件尾
result += char(0xff);
result += char(0xd9);
```

之后我们就可以具体的进行之前所说的 DCT，量化过程还有 Zigzag 排序。

DCT 过程：

```
void DCT(double f[8][8], double F[8][8]) {

    //f[8][8]是DCT前的，F[8][8]是DCT后的
    for (int i = 0; i < 8; ++i)
    {
        for (int j = 0; j < 8; j++)
        {
            F[i][j] = 0;
        }
    }

    for (int x = 0; x < 8; x++)
        for (int y = 0; y < 8; y++) {
            for (int i = 0; i < 8; i++)
            {
                for (int j = 0; j < 8; j++) {
                    F[x][y] += f[i][j]
                        * cos((2 * i + 1) / 16.0 * x * Pi)
                        * cos((2 * j + 1) / 16.0 * y * Pi);
                }
            }
            F[x][y] *= alpha(x) * alpha(y);
        }

    return;
}
```

我们使用的具体计算公式就是：

二维离散余弦变换公式为：

$$F(x,y) = alpha(x) * alpha(y) * \sum_{i=0}^{7}\sum_{j=0}^{7} f(i,j)cos(\frac{2i+1}{16}x\pi)cos(\frac{2j+1}{16}y\pi) \qquad x,y = 0,1,\ldots,7$$

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

Alpha 的实现过程就是：

```
double alpha(int u) {
    if (u == 0)
        return 1.0 / sqrt(8);
    else
        return 1.0 / 2;
}
```

然后是量化过程，量化公式是利用之前使用的两个数组进行量化：
使用的公式：

$$B_{i,j} = round\left(\frac{G_{i,j}}{Q_{i,j}}\right) \qquad i,j = 0,1,2,\ldots,7$$

```
void quantify(double YCbCr[8][8], const int Q[8][8], int* tmp) {
    for (int i = 0; i < 8; i++)
        for (int j = 0; j < 8; j++) {
            tmp[i * 8 + j] = int(round(YCbCr[i][j] / Q[i][j]));

        }
}
```

之后是 Zigzag 排序步骤，排序的形式之前已经提到了。
Zigzag 排序：

```
//对一个64位的数组进行ZigZag排序
void Zigzag_sort(int tmp[64])
{
    int ZigZag_tmp[64];
    for (int i = 0; i < 64; ++i)
    {
        ZigZag_tmp[ZIGZAG[i]] = tmp[i];
    }
    for (int i = 0; i < 64; ++i)
    {
        tmp[i] = ZigZag_tmp[i];
    }

    return;
}
```

之后就是编码的过程：
首先是原码数据，之前获得的就是原码数据。
之后我们需要将其转为 RLE 编码，BIT 编码还有 Huffman 编码：
在下面的代码中我们直接完成了这一步。

```
    //找到从后往前的最后一个0
    int last_zero = 0;
    for (int i = 63; i > 0; i--) {
        if (tmp[i] == 0)
            last_zero++;
        else
            break;
    }
```

```
for (int i = 0; i < 64 - last_zero; i++) {
    //如果当前值不为0，或者满足16个单元时
    if (i == 0 || tmp[i] != 0 || zero_num == 15) {

        //正负标志位
        int flag = 0;
        if (tmp[i] < 0)
            flag = 1, tmp[i] *= -1;

        bitset<50>bit(tmp[i]);
        int bit_len = Get_BitLen(tmp[i]);
        //加入对应的huffman编码
        tmp_str = tmp_str + Huffman_code[DC_AC_type + is_AC][(zero_num << 4) | bit_len];

        for (int i = bit_len - 1; i >= 0; i--)
        {
            if (bit[i] ^ flag)
                tmp_str = tmp_str + '1';
            else
                tmp_str = tmp_str + '0';
        }
        zero_num = 0;
    }
    else
        zero_num++;
    //第一次处理之后就可以用AC处理了
    is_AC = 1;
```

```
//如果末尾有0，加入EOB
if (last_zero)
    tmp_str = tmp_str + Huffman_code[DC_AC_type + 1][0];
```

最后就是将获得的 01 序列，将其按照 8 位划分转换为字符：
//对之前获得的 tmp_str 分隔成字节，写入结果中
//如果不足 8 位，可以在末尾补 0

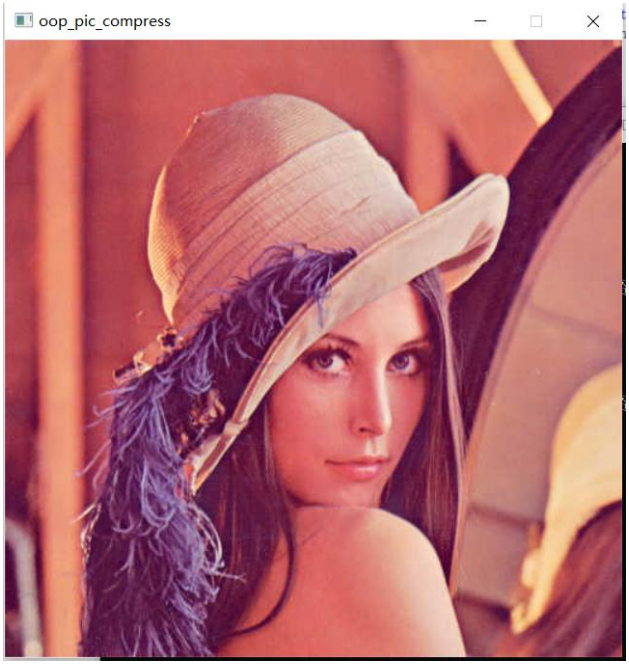
```
void write_byte()
{
    int cur_bit = 0;

    while (tmp_str.length() >= cur_bit + 8) {
        bitset<8>bit(tmp_str.substr(cur_bit, 8));
        BYTE byte = BYTE(bit.to_ullong());
        result += char(byte);
        if (byte == 0xff)
            result += char(0x00);

        cur_bit += 8;
    }
    if (cur_bit == tmp_str.length())
        return;
    else
    {

        int number = 8 - (int(tmp_str.length()) - cur_bit);
        for (int i = 0; i < number; i++)
        {
            tmp_str += '0';
        }
        bitset<8>bit(tmp_str.substr(cur_bit, 8));
        BYTE byte = BYTE(bit.to_ullong());
        result += char(byte);
        if (byte == 0xff)
            result += char(0x00);
    }

    return;
```

（6）实验结果：
结果展示：

压缩结果展示：

```
D:\桌面资料\oop_pic_compress\oop_pic_compress>D:\桌面资料\oop_pic
.tiff
开始运行
程序执行时间: 2.3342 秒
Complete!
```

读取结果展示：



x64	2023/12/1...	文件夹	
lena.jpg	2023/12/1...	JPG 图片文件	31 KB
lena.tiff	2020/5/22 ...	WPS.PIC.tiff	769 KB
main.cpp	2023/12/1...	C++ 源文件	18 KB
main.h	2022/9/5 2...	C Header 源文件	1 KB

二、问题与解决方法

1. 遇到的问题

整体上项目十分复杂，需要查找大量的资料，才能勉强搞明白其中具体的过程。
（1）使用工具的缺乏。
（2）中间处理数据太多，导致的数据混杂的问题。
（3）Jpg 的文件格式

2. 解决的方法

（1）使用工具的缺乏。

在过程中我们会涉及一个字符转 Huffman 编码的过程，这个感觉不使用 map 等等知识点，写起来十分难受。在刚开始的几天十分难受，直到群里的一个同学问到了 STL 的问题，我才知道可以使用，之后就比较顺利的解决问题了。

（2）中间处理数据太多，导致的数据混杂的问题。

在我刚开始的程序编写中，为自己定义的结构体是有 RGB[3][8][8]，有 YCbCr[3][8][8]，有 DCT_YCbCr[3][8][8]。

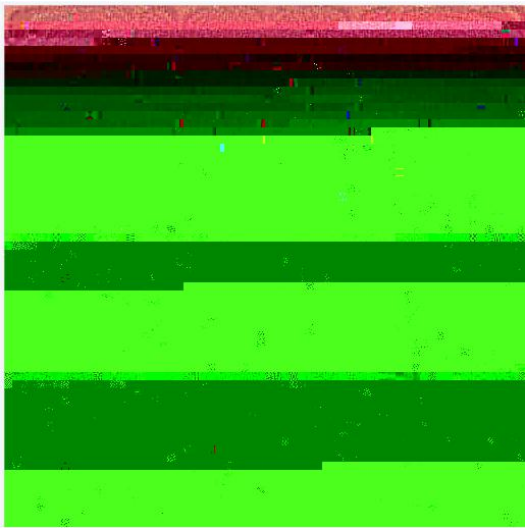
相信大家应该比较能弄明白这些变量的意义。自认为不会弄乱，结果就是绝对会弄乱。中间已经崩溃到找一些代码，然后查看具体的每个 Block 块中 DCT 处理之后的 DCT_YCbCr 值。大量的数据，大量的逐语句调试，才能勉强找到问题。

（3）jpg 的文件格式

刚开始我不知道 jpg 的文件格式，然后我们将处理的东西直接输入，发现画面太美，不敢看。知道从网上找到文章说这个东西。但是仍然不太对，后来只好向其他同学求助，才知道具体的文件格式。

说实话，格式是真多。

（4）过程大赏：



三、心得体会

这次大作业总体来说比较复杂，使用的是 JPEG 的压缩过程，这需要查询大量的资料，有的在网上还不全面。需要结合 github 上的一些代码加深理解。

但是这也十分考验自己的学习能力，在刚开始的时候十分困难，因为中间出现的不懂的地方，还有众多的 bug 都是难以解决的。这需要我们找相关的帮助软件，将转换出来的结果一一对比，才能勉强发现问题。

这次实验本就实用性，类似于一个图像压缩器。老师也提供了很多的资料，比较喜欢的是一个学长写的“压缩方法简介”，看着增长了很多见识。

希望各位助教手下留情，嘻嘻。

四、源代码

```
#include "PicReader.h"
#include <iostream>
#include <stdio.h>
#include<cmath>
#include<bitset>
#include<map>
#include<fstream>
#include<time.h>

using namespace std;

#define Pi 3.14159

//定义的 8*8 结构体块
struct compress_block {
    double R[8][8], G[8][8], B[8][8];
    double Y[8][8], Cb[8][8], Cr[8][8];
    struct compress_block* next_block;
};

//PEG 算法提供了两张标准化系数矩阵，分别处理亮度数据和色差数据
//表 1
const int Qy[8][8] = {
    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
};

//表 2
const int Qc[8][8] = {
    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
};

//ZIGZAG 排序数组
const char ZIGZAG[64] = {
    0, 1, 5, 6, 14, 15, 27, 28,
    2, 4, 7, 13, 16, 26, 29, 42,
    3, 8, 12, 17, 25, 30, 41, 43,
    9, 11, 18, 24, 31, 40, 44, 53,
    10, 19, 23, 32, 39, 45, 52, 54,
    20, 22, 33, 38, 46, 51, 55, 60,
    21, 34, 37, 47, 50, 56, 59, 61,
    35, 36, 48, 49, 57, 58, 62, 63
};

//DC，AC 编码数组
BYTE Standard_DC_Luminance_NRCodes[] = { 0, 0, 7, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0 };
BYTE Standard_DC_Luminance_Values[] = { 4, 5, 3, 2, 6, 1, 0, 7, 8, 9, 10,
11 };
BYTE Standard_DC_Chrominance_NRCodes[] = { 0, 3, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0, 0, 0 };
BYTE Standard_DC_Chrominance_Values[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11 };
BYTE Standard_AC_Luminance_NRCodes[] = { 0, 2, 1, 3, 3, 2, 4, 3, 5, 5,
4, 4, 0, 0, 1, 0x7d };
BYTE Standard_AC_Luminance_Values[] = {
    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
};
BYTE Standard_AC_Chrominance_NRCodes[] = { 0, 2, 1, 2, 4, 4, 3, 4, 7, 5,
4, 4, 0, 1, 2, 0x77 };
BYTE Standard_AC_Chrominance_Values[] = {
    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
```

```
};

//Huffman 编码表
map<unsigned char, string>Huffman_code[4];
string result; //答案字符串

//由于中间需要先转换为 Huffman 编码，再按 8 位转换为字符
//中间储存字符串
string tmp_str;
int lastDC[3];



//压缩图片函数
void Compress_pic(const char* filename);
//读图片函数
void Read_pic(const char* filename);
//将压缩过程中读到的数据 data 分块为 8*8
int div_88(compress_block* head, BYTE*& data, int x, int y);
//为每一块进行计算，算出 YCbCr
void RGB_cpu(compress_block* new_node);
//获得压缩结果字符串 result 的函数
void Get_result(int img_height, int img_width, compress_block* head);
//具体的压缩编码过程
void compress_process(compress_block* head);
//压缩编码结束后，需要将 Huffman 编码按照 8 位转换为新字符
void write_byte();

//下面详述压缩过程
//DCT 过程
void DCT(double f[8][8], double F[8][8]);
//DCT 中用到 alpha 函数
double alpha(int u);
//数据量化过程
void quantify(double YCbCr[8][8], const int Q[8][8], int* tmp);
//Zigzag 排序过程
void Zigzag_sort(int tmp[64]);
//获得具体编码过程
void writeCode(int* tmp, int DC_AC_type, int num);

//初始化 Huffman 编码
void Init_Huffman_Code();
//根据 DC，AC 系数表获得 Huffman 编码
void Get_Huffman_Code(BYTE* NRCodes, BYTE* Values, int index);
```

```
int Get_BitLen(int bit);
//动态内存申请之后的释放函数
void delete_block(compress_block* head);



double alpha(int u) {
    if (u == 0)
        return 1.0 / sqrt(8);
    else
        return 1.0 / 2;
}

//离散余弦变换（DCT）
void DCT(double f[8][8], double F[8][8]) {

    //f[8][8]是DCT前的，F[8][8]是DCT后的
    for (int i = 0; i < 8; ++i)
    {
        for (int j = 0; j < 8; j++)
        {
            F[i][j] = 0;
        }
    }

    for (int x = 0; x < 8; x++)
        for (int y = 0; y < 8; y++) {
            for (int i = 0; i < 8; i++)
            {
                for (int j = 0; j < 8; j++) {
                    F[x][y] += f[i][j]
                        * cos((2 * i + 1) / 16.0 * x * Pi)
                        * cos((2 * j + 1) / 16.0 * y * Pi);
                }
            }
            F[x][y] *= alpha(x) * alpha(y);
        }

    return;
}

void quantify(double YCbCr[8][8], const int Q[8][8], int* tmp) {
    for (int i = 0; i < 8; i++)
```

```
        for (int j = 0; j < 8; j++) {
            tmp[i * 8 + j] = int(round(YCbCr[i][j] / Q[i][j]));

        }
}

//对一个 64 位的数组进行 ZigZag 排序
void Zigzag_sort(int tmp[64])
{
    int ZigZag_tmp[64];
    for (int i = 0; i < 64; ++i)
    {
        ZigZag_tmp[ZIGZAG[i]] = tmp[i];
    }
    for (int i = 0; i < 64; ++i)
    {
        tmp[i] = ZigZag_tmp[i];
    }

    return;
}

//求整数 bit 的二进制长度
int Get_BitLen(int bit) {
    //如果是 0 直接返回 0
    if (bit == 0)
        return 0;

    //如果为负数，需要转为正数
    if (bit < 0)
        bit = -bit;

    int length = 0;
    while (bit > 0) {
        length++;
        bit >>= 1;
    }
    return length;
}

void writeCode(int* tmp, int DC_AC_type, int num) {
    //对第一个值（DC）进行特殊处理：减去前一个小块该值的原值
    int tmp_1 = tmp[0];
    tmp[0] -= lastDC[num];
```

```
    lastDC[num] = tmp_1;

    //找到从后往前的最后一个 0
    int last_zero = 0;
    for (int i = 63; i > 0; i--) {
        if (tmp[i] == 0)
            last_zero++;
        else
            break;
    }

    //前面需要统计 0 的个数
    int zero_num = 0;
    //第一个用 DC 处理，之后用 AC 进行处理
    int is_AC = 0;

    for (int i = 0; i < 64 - last_zero; i++) {
        //如果当前值不为 0，或者满足 16 个单元时
        if (i == 0 || tmp[i] != 0 || zero_num == 15) {

            //正负标志位
            int flag = 0;
            if (tmp[i] < 0)
                flag = 1, tmp[i] *= -1;

            bitset<50>bit(tmp[i]);
            int bit_len = Get_BitLen(tmp[i]);
            //加入对应的 huffman 编码
            tmp_str    =    tmp_str    +    Huffman_code[DC_AC_type    +
is_AC][(zero_num << 4) | bit_len];

            for (int i = bit_len - 1; i >= 0; i--)
            {
                if (bit[i] ^ flag)
                    tmp_str = tmp_str + '1';
                else
                    tmp_str = tmp_str + '0';
            }
            zero_num = 0;
        }
        else
            zero_num++;
        //第一次处理之后就可以用 AC 处理了
        is_AC = 1;
```

```
    }

    //如果末尾有0，加入EOB
    if (last_zero)
        tmp_str = tmp_str + Huffman_code[DC_AC_type + 1][0];

    return;
}

void Get_Huffman_Code(BYTE* NRCodes, BYTE* Values, int index) {

    BYTE* values_ptr = Values;
    int code = 0;
    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < int(NRCodes[i]); j++) {
            BYTE values_value = *values_ptr;
            string str;
            bitset<50>bit(code);
            for (int k = i; k >= 0; k--) {
                if (bit[k] == 0)
                    str = str + '0';
                else
                    str = str + '1';
            }
            Huffman_code[index][values_value] = str;

            values_ptr++;
            code++;
        }
        code <<= 1;
    }
}

void Init_Huffman_Code() {
    Get_Huffman_Code(Standard_DC_Luminance_NRCodes,
Standard_DC_Luminance_Values, 0);
    Get_Huffman_Code(Standard_AC_Luminance_NRCodes,
Standard_AC_Luminance_Values, 1);
    Get_Huffman_Code(Standard_DC_Chrominance_NRCodes,
Standard_DC_Chrominance_Values, 2);
    Get_Huffman_Code(Standard_AC_Chrominance_NRCodes,
Standard_AC_Chrominance_Values, 3);
}
```

```
void compress_process(compress_block* head) {

    //初始化哈夫曼编码
    Init_Huffman_Code();

    compress_block* curnode;
    curnode = head;

    while (curnode->next_block) {
        //获得当前要处理的 8*8 块
        curnode = curnode->next_block;

        //用于储存 DCT 之后的数据
        double F[8][8];
        int tmp[64];

        //之后对 8*8 块的 Y，Cb，Cr 都要进行处理
        //对 Y 进行处理
        DCT(curnode->Y, F);
        for (int i = 0; i < 8; ++i)
        {
            for (int j = 0; j < 8; j++)
            {
                curnode->Y[i][j] = F[i][j];
            }
        }


        quantify(curnode->Y, Qy, tmp);
        Zigzag_sort(tmp);
        writeCode(tmp, 0, 0);




        //对 Cb 进行处理
        DCT(curnode->Cb, F);
        for (int i = 0; i < 8; ++i)
        {
            for (int j = 0; j < 8; j++)
            {
                curnode->Cb[i][j] = F[i][j];
            }
        }
```

```
        quantify(curnode->Cb, Qc, tmp);
        Zigzag_sort(tmp);
        writeCode(tmp, 2, 1);

        //对 Cr 进行处理
        DCT(curnode->Cr, F);
        for (int i = 0; i < 8; ++i)
        {
            for (int j = 0; j < 8; j++)
            {
                curnode->Cr[i][j] = F[i][j];
            }
        }

        quantify(curnode->Cr, Qc, tmp);
        Zigzag_sort(tmp);
        writeCode(tmp, 2, 2);

    }
    return;
}

//对之前获得的 tmp_str 分隔成字节，写入结果中
//如果不足 8 位，可以
void write_byte()
{
    int cur_bit = 0;

    while (tmp_str.length() >= cur_bit + 8) {
        bitset<8>bit(tmp_str.substr(cur_bit, 8));
        BYTE byte = BYTE(bit.to_ullong());
        result += char(byte);
        if (byte == 0xff)
            result += char(0x00);

        cur_bit += 8;
    }

    if (cur_bit == tmp_str.length())
        return;
    else
    {
```

```
            int number = 8 - (int(tmp_str.length()) - cur_bit);
            for (int i = 0; i < number; i++)
            {
                tmp_str += '0';
            }
            bitset<8>bit(tmp_str.substr(cur_bit, 8));
            BYTE byte = BYTE(bit.to_ullong());
            result += char(byte);
            if (byte == 0xff)
                result += char(0x00);
        }

        return;
    }

    void Get_result(int img_height, int img_width, compress_block* head) {

        //SOI 文件头
        result += char(0xff);
        result += char(0xd8);

        //APP0 图像识别信息
        //图像识别信息头
        result += char(0xff);
        result += char(0xe0);
        //段长度
        result += char(0x00);
        result += char(0x10);
        //交换格式
        result += char(0x4A);
        result += char(0x46);
        result += char(0x49);
        result += char(0x46);
        result += char(0x00);
        //主版本号和次版本号
        result += char(0x01);
        result += char(0x02); //不一样
        //单位密度
        result += char(0x00); //不一样
        //X 像素密度
        result += char(0x00); //不一样
        result += char(0x01);
        //Y 像素密度
        result += char(0x00); //不一样
```

```
result += char(0x01);
//缩略图 X 像素
result += char(0x00);
//缩略图 Y 像素
result += char(0x00);
//RGB 缩略图
//这里是空的

//DQT 定义量化表
//定义量化表的头
result += char(0xff);
result += char(0xdb);
//段长度
result += char(0x00);
result += char(0x84);
//QT 信息
result += char(0x00);
//QT 量化表 量化表 0
int Zigzag_1[64], Zigzag_2[64];
for (int i = 0; i < 8; i++)
    for (int j = 0; j < 8; j++)
        Zigzag_1[i * 8 + j] = Qy[i][j];
for (int i = 0; i < 64; i++)
    Zigzag_2[ZIGZAG[i]] = Zigzag_1[i];
for (int i = 0; i < 64; i++)
    result += char(Zigzag_2[i]);

//QT 信息
result += char(0x01);
//QT 量化表 量化表 1
for (int i = 0; i < 8; i++)
    for (int j = 0; j < 8; j++)
        Zigzag_1[i * 8 + j] = Qc[i][j];
for (int i = 0; i < 64; i++)
    Zigzag_2[ZIGZAG[i]] = Zigzag_1[i];
for (int i = 0; i < 64; i++)
    result += char(Zigzag_2[i]);

//SOF0 图像基本信息
//图像基本信息
result += char(0xff);
result += char(0xc0);
//段长度
result += char(0x00);
```

```
result += char(0x11);
//样本精度
result += char(0x08);
//样本高度
result += char(img_height >> 8);
result += char(img_height & 0b1111);
//样本宽度
result += char(img_width >> 8);
result += char(img_width & 0b1111);
//组件数量
result += char(0x03);
//Y 组件
result += char(0x01); //组件 ID
result += char(0x11); //采样系数
result += char(0x00); //量化表号
//Cb 组件
result += char(0x02);
result += char(0x11);
result += char(0x01);
//Cr 组件
result += char(0x03);
result += char(0x11);
result += char(0x01);

//DHT 定义 huffman 表
//Huffman 表头
result += char(0xff);
result += char(0xc4);
//段长度
result += char(0x01);
result += char(0xa2);
//HT 信息
//DC HT 为 0
result += char(0x00);
for (int i = 0; i < 16; i++)
    result += char(Standard_DC_Luminance_NRCodes[i]);
for (int i = 0; i < 12; i++)
    result += char(Standard_DC_Luminance_Values[i]);
//AC HT 为 0
result += char(0x10);
for (int i = 0; i < 16; i++)
    result += char(Standard_AC_Luminance_NRCodes[i]);
for (int i = 0; i < 162; i++)
    result += char(Standard_AC_Luminance_Values[i]);
```

```
    //DC HT 为 1
    result += char(0x01);
    for (int i = 0; i < 16; i++)
        result += char(Standard_DC_Chrominance_NRCodes[i]);
    for (int i = 0; i < 12; i++)
        result += char(Standard_DC_Chrominance_Values[i]);
    //AC HT 为 1
    result += char(0x11);
    for (int i = 0; i < 16; i++)
        result += char(Standard_AC_Chrominance_NRCodes[i]);
    for (int i = 0; i < 162; i++)
        result += char(Standard_AC_Chrominance_Values[i]);

    //SOS 扫描行开始
    //扫描行开始的头
    result += char(0xff);
    result += char(0xda);
    //段长度
    result += char(0x00);
    result += char(0x0c);
    //扫描行内组件数量
    result += char(0x03);
    //Y 分量
    //第一个字节是组件 ID；第二个字节 0-3 位 AC 表号，4-7 位 DC 表号，表号
的值是 0-3。
    result += char(0x01);
    result += char(0x00);
    //Cb 分量
    result += char(0x02);
    result += char(0x11);
    //Cr 分量
    result += char(0x03);
    result += char(0x11);
    //剩余 3 个字节,用途不明,忽略
    result += char(0x00);
    result += char(0x3f);
    result += char(0x00);



    compress_process(head);

    write_byte();
```

```
    //EOI 文件尾
    result += char(0xff);
    result += char(0xd9);

    return;
}

void RGB_cpu(compress_block* new_node)
{
    for (int i = 0; i < 8; ++i)
    {
        for (int j = 0; j < 8; ++j)
        {
            new_node->Y[i][j] = 0.29871 * new_node->R[i][j] + 0.58661 *
new_node->G[i][j] + 0.11448 * new_node->B[i][j] - 128;
            new_node->Cb[i][j] = -0.16874 * new_node->R[i][j] - 0.33126
* new_node->G[i][j] + 0.50000 * new_node->B[i][j];
            new_node->Cr[i][j] = 0.50000 * new_node->R[i][j] - 0.41869 *
new_node->G[i][j] - 0.08131 * new_node->B[i][j];
        }
    }

    return;
}

//将所有的数据划分为 8*8 的块的链表
int div_88(compress_block* head, BYTE*& data, int x, int y)
{
    compress_block* curnode;

    //数据转换成 8*8 的整块
    for (int i = 0; i < x; i += 8)
    {
        for (int j = 0; j < y; j += 8)
        {
            compress_block* new_node = new compress_block();
            if (!new_node)
            {
                cout << "动态内存申请失败" << endl;
                return -1;
            }

            //将这一 8*8 块的 RGB 赋值进结构体中
            for (int p = i; p < i + 8; ++p)
```

```
            {
                for (int q = j; q < j + 8; ++q)
                {
                    int index = p * y * 4 + q * 4;
                    new_node->R[p - i][q - j] = data[index];
                    new_node->G[p - i][q - j] = data[index + 1];
                    new_node->B[p - i][q - j] = data[index + 2];

                }
            }
            RGB_cpu(new_node);

            curnode = head;

            //找到列表尾
            while (curnode->next_block)
            {
                curnode = curnode->next_block;
            }

            curnode->next_block = new_node;
        }
    }

    return 0;
}

void delete_block(compress_block* head) {
    compress_block* deletenode = head;
    compress_block* nextnode = head->next_block;
    while (nextnode)
    {
        deletenode->next_block = NULL;
        delete deletenode;
        deletenode = nextnode;
        nextnode = nextnode->next_block;
    }
    delete deletenode;

    return;
}

void Compress_pic(const char* filename) {
```

```
LARGE_INTEGER frequency;
LARGE_INTEGER start;
LARGE_INTEGER end;

// 获取时钟频率
QueryPerformanceFrequency(&frequency);

// 获取开始时间
QueryPerformanceCounter(&start);

PicReader imread;
BYTE* data = nullptr;
UINT x, y;
imread.readPic(filename);
imread.getData(data, x, y);

compress_block* head = new compress_block();
if (head == NULL) {
    cout << "内存申请失败" << endl;
    exit(0);
}
head->next_block = NULL;

//全部数据划分为 8*8 的块链表
int a = div_88(head, data, x, y);

delete[] data;
data = nullptr;

Get_result(x, y, head);

//输出
const char* outfile = "lena.jpg";
ofstream fl(outfile, ios::binary);
if (!fl) {
    cout << "文件打开失败" << endl;
    exit(0);
}
fl << result;
fl.close();

//释放动态申请的空间（链表）
delete_block(head);
```

```
    // 获取结束时间
    QueryPerformanceCounter(&end);

    // 计算程序执行时间
    double duration = static_cast<double>(end.QuadPart - start.QuadPart)
/ frequency.QuadPart;

    // 输出程序执行时间（以秒为单位）
    cout << "程序执行时间: " << duration << " 秒" << endl;

    return;
}

void Read_pic(const char* filename) {

    PicReader imread;
    BYTE* data = nullptr;
    UINT x, y;
    imread.readPic(filename);
    imread.getData(data, x, y);
    imread.showPic(data, x, y);

    delete[] data;
    data = nullptr;

    printf("Press enter to continue...");
    (void)getchar();
}

//主函数
int main(int argc, char** argv) {
    cout << "开始运行" << endl;

    if (argc != 3) {
        cerr << "请检查参数个数是否正确" << endl;
        return -1;
    }

    if (!strcmp(argv[1], "-compress") || !strcmp(argv[1], "-read")) {
        if (!strcmp(argv[1], "-compress"))
        {
            Compress_pic(argv[2]);
        }
        else if (!strcmp(argv[1], "-read"))
```

```
        {
            Read_pic(argv[2]);
        }
    }
    else {
        cerr << "Unknown parameter!\nCommand list:\n-compress\n-read" <<
endl;
        return -1;
    }

    cout << "Complete!" << endl;
    return 0;
}
```