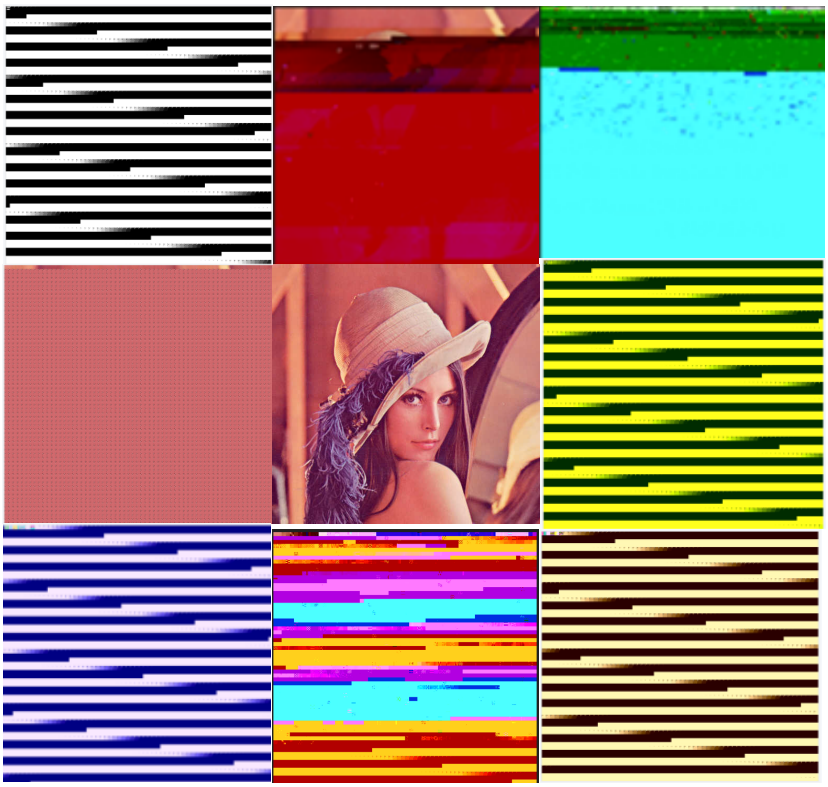
同济大学

课程作业实验报告



题目 图像压缩小程序实验报告

课 程 面向对象的程序设计

姓 名 蔡宇轩

学 号 2252429

学 院 电子信息与工程学院

专 业 计算机科学与技术22级1班

教 师 陈宇飞

二〇二三 年 十 二 月 十 五 日

1 设计思路

1.1 各文件功能分配

1.1.1 Image_main.cpp

本次作业要求使用带参主函数和类封装的方式实现。本程序命令行分为“-read”和“-compress”，对应读取并压缩输出至同一根目录下和解压缩并在弹窗中输出展示。

1.1.2 Compress.h

在本头文件中定义了一个名为JPEG的class类，用于读取原图lena.tiff并进行相关的编码压缩操作。本文件中含有JPEG的类声明和各成员函数的声明。

1.1.3 Decode.h

在本头文件中定义了一个名为Decode的class类，用于读取已经压缩好的lena.jpg并进行相关的解码解压操作。本文件中含有Decode的类声明和各成员函数的声明。

1.1.4 Image_tools.cpp

本源文件中包含了在上述两个.h文件中声明的所有类成员函数的具体实现。

1.1.5 PicReader框架

已经提供的PicReader框架，包含头文件和相关具体操作的源文件，用于读取原图lena.tiff和展示解压文件。

图 1.1-1 各工程文件的组织分配

1.2 总体设计思路

1.2.1 JPEG图片的原理

JPEG图片的本质原理在于线性代数里的基变换知识，即对于同一个线性空间（也就是此处的一张图片里的所有像素点构成的向量所组成的空间），可以由不同的基（可以理解为不同的组成单元）张成。由于同一线性空间的基之间可以相互转换（依靠基变换矩阵），因此如果我们定义一组标准的“基”，那么理论上所有图片都可以由这组基张成得到。显然由于图片大小不同，这一组基的大小将会依据图片的长宽大小而变化。因此，我们选择最常见的8*8矩阵作为一组基构成的向量，而其余大小的图片只需要看成多个8*8块矩阵的分割即可。

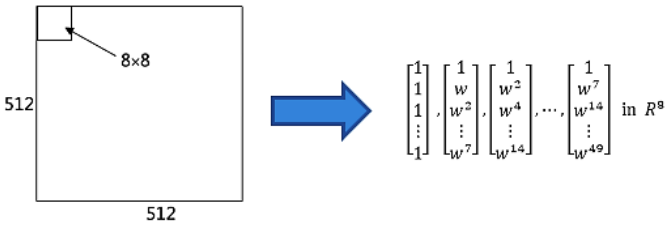


图 1.2-1 JPEG图片压缩时的分割处理

因此，根据基变换，任何一个8*8的含有64个像素点的像素矩阵，都可以表示成如下一组基的线性组合（图中每一个像素块都是8*8矩阵，是标准基里的一个单元）：

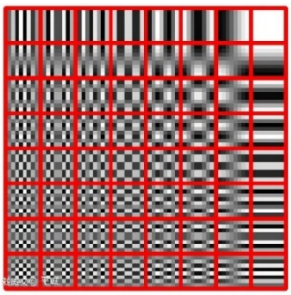


图 1.2-2 JPEG图片选取的标准基

依据线性代数的知识我们可以知道，对于两组不同的基，我们可以通过基变换矩阵建立二者之间的关系：

$$(\beta_1, \beta_2, \dots, \beta_n) = (\alpha_1, \alpha_2, \dots, \alpha_n) \cdot C$$

因此我们只需要求得系数矩阵（也就是基变换中的基变换矩阵）就能够以上述基（图1.2-2）为标准基，唯一的表示任何一张8*8的像素图片。

1.2.2 图片压缩思路

我们按照JPEG格式图片的标准编码和压缩过程，可以将一个RGB三通道的原始图片压缩至很小的大小。其压缩的过程可以主要简述如下，具体的实现过程详见第2节。

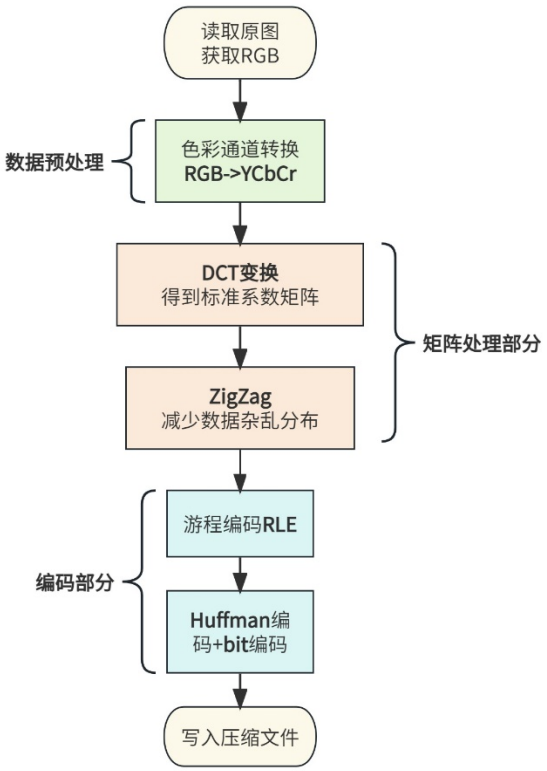


图 1.2-3 JPEG图片压缩的思路

1.2.3 图片解压思路

与压缩的思路类似，解压的本质就是将我们在上一步压缩得到压缩图片文件通过解码和相关的逆向操作，还原出原本的RGB三通道，并加载到PicReader框架下的showPic函数即可。其解压的过程可以主要简述如下，，具体的实现过程详见第2节。

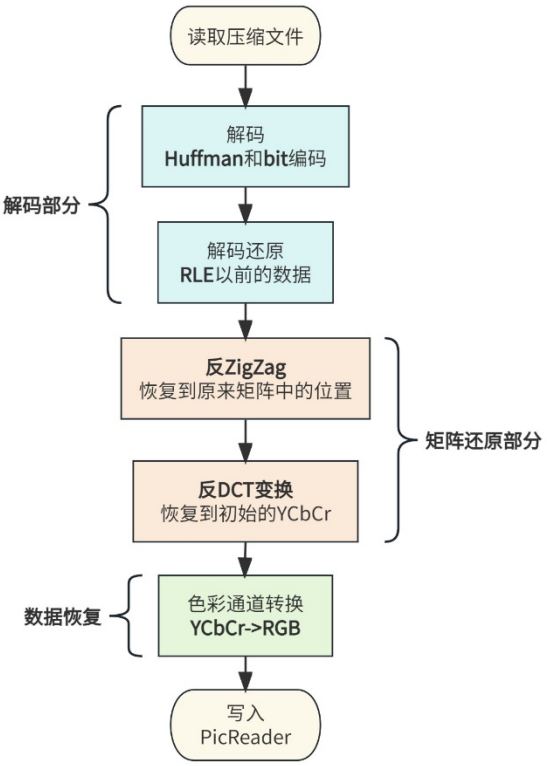


图 1.2-4 JPEG图片解压的思路

2 函数、类功能描述

2.1 类成员函数清单

2.1.1 JPEG类成员函数

```
private:
    //得到一个数字 code 的第 length 位的值
    int GetBit(unsigned int code, int length);
    //写入一个字长的数据时需要高低位互换
    void write_word(int word,ofstream&
outfile);
    //打表，写入文件头
    void writeJPEGhead(ofstream& outfile);
    //打表，写入文件尾
    void writeJPEGtail(ofstream& outfile);
    //色彩通道转换，从 rgba 转换为 Ycbcr
    void cgto_YCbCr(BYTE* RGBbuffer,double
Ydata[],double Cbdata[],double Crdata[]);
    //傅里叶 DCT 变换、量化处理并 zigzag 到一维数组
里
    void cgto_DCT(double data[], int
dct_data[], const unsigned char QuantTable[]);
    //建立四张 ACDC 表的哈夫曼编码表
    void BuildHuffTable();
    //DC 编码过程
    void DC_bitcnt(int TABLEFLAG, int
dct_z_data, Code HuffCode[], int& index,int
PREFLAG);
    //AC 编码过程
    void AC_bitcnt(int TABLEFLAG, int eob, int
dct_z_data[], Code* HuffCode, int& index);
    //Huffman 编码
    int Huff_encoding(int flagtable,int
dct_data[],Code HuffCode[],int PREFLAG);
    //写入比特流文件
    void write_bits(Code HuffCode[], const int
index);
```

```
public:
    //构造函数
    JPEG(BYTE* readData,UINT readWidth,UINT
readHeight,char readName[]) {
        img_width = readWidth;
        img_height = readHeight;
        data = readData;
        Picname = readName;
    }
    //析构函数
    ~JPEG(){};
    //压缩函数
    void MyCompress();
```

2.1.2 Decode类成员函数

```
private:
    //读取文件头
    bool readJPEGhead();
    //依据文件头的信息重建哈夫曼编码的 ACDC 表
    void readHuffTable(int tag);
    //重建哈夫曼编码表和对应的哈希表
    void BuildHuff_and_Hash();
    //读一个 8*8 矩阵对应的编码
    void readLine(int FLAG,int zdata[],int
preFlag);
    //反 zigzag、反 DCT
    void reverseDCT(int FLAG,int zdata[],
double YCbCrT[]);
    //转换回 GRB 三通道
    void cgto_RGB(double Y[], double Cb[],
double Cr[],int x,int y);
    //向下通过查询哈希表读取一个 Code
    int GetCode(int flag);
    //在文件中继续向下读取 1 个比特位
    int read_next_bit();
    //将从文件中读到的高低位倒序的数字正序计算出来
并返回
    int cgto_num(short word);
    //将计算出的 code 值转换为 string 类
    string code_to_str(int code,int code_len);
```

```
public:
    //解压函数
    void MyDecode();
    //构造函数
    Decode() {};
    //析构函数
    ~Decode() {};
```

2.2 类的主要数据成员说明

2.2.1 编码结构体

我们知道任何一个值的编码都是二进制码，即由0和1组成的字符串。为了节省存储空间，我们可以将这串二进制码对应的十进制数字和编码的长度组成一个结构体，用来表示每个编码：

```
struct Code{
    int code;//二进制码对应的十进制数
    int length;//本编码的码长
}
```

这个表示方法是本程序的关键，后续的编码、解码操作都是基于这个结构体类型完成的。

2.2.2 ACDC编码表

根据JPEG标准范式哈夫曼编码表，我们可以得到有关每一个8*8矩阵的直流（DC）与交流（AC）分量的NRCodes和Values表，即亮度Y(Luminance)和红蓝差值CbCr(Chrominance)各有AC,DC表一个。这两个表分别表示了不同码长的码数量、每一个编码对应的实际值。

```
/*********YCbCr 的 ACDC 表***********/
//Y-DC
const unsigned char DC_Y_NRCodes[16] =
{ 0,1,5,1,1,1,1,1,1 };
const unsigned char DC_Y_Values[16] =
{ 0,1,2,3,4,5,6,7,8,9,10,11 };
//C-DC
const unsigned char DC_C_NRCodes[16] =
{ 0,3,1,1,1,1,1,1,1,1,1 };
const unsigned char DC_C_Values[16] =
{ 0,1,2,3,4,5,6,7,8,9,10,11 };
//Y-AC
const unsigned char AC_Y_NRCodes[16] =
{ 0,2,1,3,3,2,4,3,5,5,4,4,0,0,1,125 };
const unsigned char AC_Y_Values[256];
    //太多，省略不写
//C-AC
const unsigned char AC_C_NRCodes[16] =
{ 0,2,1,2,4,4,3,4,7,5,4,4,0,1,2,119 };
const unsigned char AC_C_Values[256];
    //太多，省略不写
```

依据范式哈夫曼编码的建立规则，我们可以在不建立哈夫曼树的情况下，仅通过这四张表还原出JPEG格式文件的范式哈夫曼编码。

2.3 主要函数执行说明

2.3.1 色彩通道转换

编码压缩时需要将RGB转换为YCbCr,解码解压时需要将YCbCr还原回RGB。为了达到效果，我们利用了如下公式，并对参数和精度作了适当调整：

$$\begin{cases} Y = 0.29871*R + 0.58661*G + 0.11448*B-128 \\ Cb = -0.16874*R-0.33126*G + 0.5*B \\ Cr = 0.5*R-0.41869*G-0.08131*B \end{cases}$$

$$\begin{cases} R = 1.0002 * (Y + 128) - 0.000992 * Cb + 1.402126 * Cr \\ G = 1.0002*(Y + 128)-0.345118*Cb-0.714010*Cr \\ B = 1.0002*(Y + 128) + 1.771018*Cb + 0.000143*Cr \end{cases}$$

2.3.2 DCT与反DCT变换

傅里叶正变换（DCT）和逆变换（反DCT）本质上就是求基变换系数矩阵的过程，我们可以直接利用傅里叶变换的公式，利用循环语句实现即可。

DCT变换：

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

$$alpha(x) = \begin{cases} 1/\sqrt{8}, x = 0 \\ 1/2, x \neq 0 \end{cases}$$

反DCT变换：

$$f(i,j) = \sum_{u=0}^{7}\sum_{v=0}^{7} c(u)c(v)F(u,v)cos\left[\frac{2i+1}{16}u\pi\right]cos\left[\frac{2j+1}{16}v\pi\right]$$

$$c(u) = \begin{cases} 1/\sqrt{8}, u = 0 \\ 1/2, u \neq 0 \end{cases}$$

2.3.3 创建哈夫曼编码

范式哈夫曼编码有如下规则：

（1）相同码长的编码，相邻之间只有1位的差距，本质是格雷码

（2）码长+1对应编码整体左移1位

我们以i表示码长，以j表示当前码长下的第j个编码，则可以得到有关JPEG范式哈夫曼编码的数学规则如下：

$$c[i]_{j+1} = c[i]_j + 1$$

$$c[i+1] = c[i] \ll 1 = c[i] * 2$$

因此，我们可以得到依据AC表和DC表生成范式哈夫曼编码表的算法，每次我们都可以算的编码的长度和编码对应的十进

制数字，构成新的结构体变量，存入Code数组即可。

```
//依据 AC 和 DC 表复现范式哈夫曼编码表的算法伪代码
for(type=0;type<4;type++){
    index;//在 Code 数组表中的下标索引
    code;//编码对应的十进制数字
   for(int i=0;i<sizeof(NRCodes[]);i++){
       for(int j-0;j<NRCodes[i];j++){
        //没有长度为 0 的编码，码长从 1 开始
        //范式哈夫曼：同一长度的码长相邻只差 1
        HuffTable[AC[type][index]].code=code;
        HuffTable[AC[type][index]].length=i+1;
            code++;
            index++;
        }
        code <<= 1;//相邻码长：
        //某长度开始的第一个编码等于上一个长度的最后一个编码左移一位
    }
}//BuildHuffman
```

2.3.4 压缩编码

对于RLE游程编码以后的数组，我们只记录非零数字和其之前的连续0数量，因此对于这样一个数组：

$$16,1,1,0,0,0,0,0,\ldots\ldots,0$$

我们就可以将其游程编码为：

$$(0,16)(0,1)(0,1)EOB$$

其中第一个分量叫做DC分量，在编码时要调用相应的DC哈夫曼编码表，后面的都是AC分量，调用AC哈夫曼编码表。

我们求得非零数字的二进制表示，其二进制码的长度即为这个数值在bit编码表中的行数；若这个数值是正数，则其bit编码就是正数的二进制表示本身，若为负数，则将正数的码按位取反，得到负数的bit码。

我们将连续0的个数zerocnt与当前非0值在bit表中的行数bit_len组成一个8位字符型，这个值就是我们在哈夫曼编码表中对应的Value，只需要查表获得这个value对应的编码就可以。后面的bit编码直接保留即可。

```
//huff_encoding 算法伪代码
while(i<EOB){
    count zerocnt;
    //counting num of zero
    if(FZR)
        /*存储连续 16 个 0*/
    else{
        abs_val;//求当前值的绝对值
        count bit_len;
        Code
Huffman=HuffTable[zerocnt*16+bit_len];
     //哈夫曼编码表是按照 value 作为数组下标索引的
        if(positive)
            Code BitCode={abs_val,bit_len}
        else//negative
            Code BitCode={~abs_val,bit_len};
    }
}
//注意：相应的数组下标和计数量清零等操作此处未写
```

在写解压文件时，依旧按照8个 bit位拼出1个char的方式写入文件即可。

2.3.5 解压解码

解码就是上次的逆过程，我们每次向下读取一个码，并按照重建的哈夫曼编码的哈希表查询当前的码对应的值，然后分解出zerocnt和bit_len。根据bit_len向下读取一定长度的码值，并判断实际非零数字的正负。将计算得到的值都依次加入我们的还原数组即可。

由于解码就是2.3.4的逆向过程，此处不再作详细阐释。

3 主程序流程图

3.1 主体框架简述

本作业要求使用带参主函数和命令行的形式调用相关功能。主函数判断命令行参数的数量即输入合法性，然后依据“-read”和“-compress”分别调用解压和压缩功能，并输出相关的个性化提示。

3.2 主函数流程图

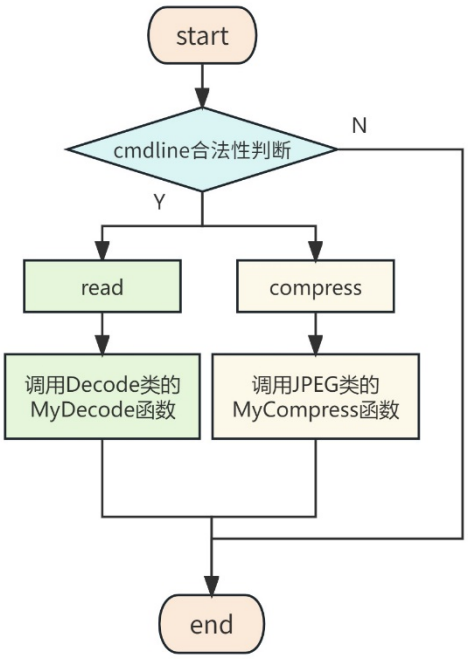


图 3.2-1 主函数流程图

4 实验过程遇到的问题及解决

如若说起写Lena写出多少种错误来，那便是没个几天几夜也说不完了。从我的诸多bug中挑选了几个比较致命和不易察觉的，在这里简记一二。

4.1 写入字长的数据高低位逆置

写入16位数据，即一个字长的数据时，需要将高低位逆置：先将高8位写入压缩文件，然后再将低8位写入文件中。

```
void JPEG::write_word(int word,ofstream&outfile) {
    char low = (char)word;
    char high = (char)(word >> 8);
    outfile << high << low;
}
```

图 4.1-1 字长数据高低位逆置

同样的问题，在解码的时候同样遇到，读到压缩图片的大小为2*2： 为什么呢？原图图像宽度和高度是512：二进制是00000010 00000000，但是在写的时候是先写高位再写低位，所以直接读的时候，读到的就是00000000 00000010=2。

```
//将从文件中读到的高低位倒序的数字正序计算出来并返回
int Decode::cgto_num(short word) {
    int num_low= word >> 8;
    int num_high = (int)((char)word);
    int num = num_high << 8 + num_low;
    return num;
}
```

图 4.1-2 字长数据高低位矫正

添加顺序矫正的函数，修改正确：

```
img_height = (UINT)cgto_num(word);
cout << "读取到的图像高度="<<img_height;
getchar();

infile.read
img_width =
cout << "读
```

```
d:\Desktop\ImageCompress\Debug\ImageCompress.
您正在使用PicReader框架读取并展示
读取到的图像高度=512
读取到的图像宽度=512
```

图 4.1-3 修改后能正确读取图像大小

4.2 再窥文件打开方式

在打文件头时，发现0a前总是多跟一个0d，但是直接在控制台输出这个0d又消失，想了好久百思不得其解：

```
 8  9  a  b  c  d  e  f  Dump
49 46 00 01 01 00 00 01   ??.JFIF
00 10 0b 0c 0e 0c 0d 0a  ....  ??.
选择 D:\Desktop\ImageCompress\Debug\ImageCompr
10 0b 0c 0e 0c 0a 10 0e 0d 0e 12 1
67 68 67 3e 4d 71 79 70 64 78 5c 6
```

图 4.2-1 文件写入与控制台输出不一致

还是因为这个十六进制看不出来，转换成对应的字符型才知道这对应的是\r\n！因此一定是文本模式和二进制模式的区别。原来是我打开文件的时候没有加“ios::binary”，加上之后问题解决。

4.3 矩阵的计算反复出错

虽然一次计算64个像素，但是这64个像素在图像上表现为1个8*8矩阵，因此他们在BYTE* data数组中并不连续。所以，想要正确找到数组下标索引并赋值，就需要使用2层for循环。

```
//index是相对于RGBbuffer的偏移量
int index = 0;
for (int i = 0; i < 8; i++) {
    int first = index;
    for (int j = 0; j < 8; j++) {
        double R = (double)RGBbuffer[index], G = (double)RGBbuffer[index + 1], B = (double)RGBbuffer[index + 2];
        Ydata[i * 8 + j] = 0.299 * R + 0.587 * G + 0.114 * B - 128;
        Cbdata[i * 8 + j] = -0.1687 * R - 0.3313 * G + 0.5 * B;
        Crdata[i * 8 + j] = 0.5 * R - 0.4187 * G - 0.0813 * B;
        index += 4;
    }
    index = first + img_width * 4;
}
```

图 4.3-1 矩阵的位置计算有误后的修正

无独有偶，在解码的时候也遇到了这个下标索引计算有误的问题：由于忘记乘上4（RGBA一共4个通道），所以最后showPic出了4个Lena：

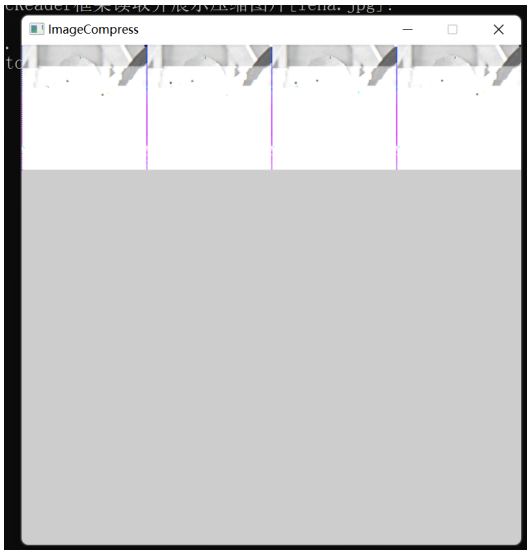


图 4.3-2 4个lena把我吓死了

把矩阵反复打印出来比对，终于改对。

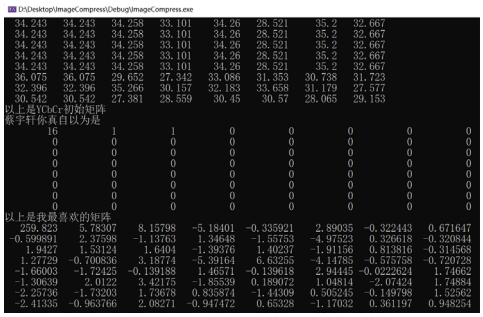


图 4.3-3 矩阵的位置计算有误后的修正

4.4 万恶的哈夫曼编码

在这之前打出了太多斑马病，心态几近崩溃：

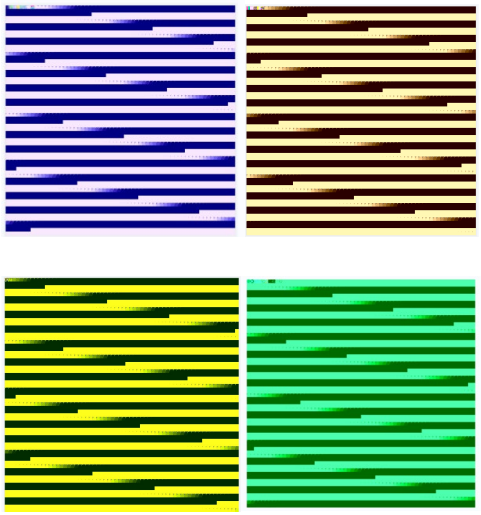


图 4.4-1 多彩斑马

肉眼实在查不出错误，于是开始单步调试，没执行一次编码就把相关内容都打印出来，一步一步getchar看：

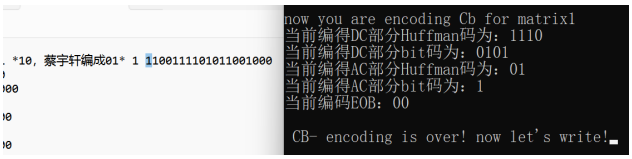


图 4.4-2 单步执行调试编码过程

发现是AC编码有问题，上图发生的问题就是第一个矩阵的Cb矩阵，-10后面的那个0没有计进去，走到1的时候zerocnt仍为0：

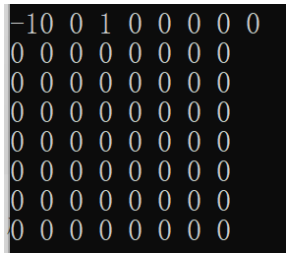


图 4.4-3 AC编码存在问题

最后查得是AC_bitcnt函数计算第一个合并字符的时候出现了问题，应该加上abs_val（即原始数据的绝对值）的长度而不是原始数据！如果是负数，那么就无法在赫夫曼编码表里索引到编码了，因此出现了大量的斑马条纹。

```
//首先编哈夫曼码
unsigned char combine = zerocnt * 16 + dct_z_data[i];
HuffCode[index] = *(AC_table[TABLEFLAG] + combine);
index++;
```

图 4.4-4 AC编码的索引存在问题

4.5 比特计数位的玄机

在编码的时候就遇到跳位写入，有些比特位写不进压缩文件的情况，其原因是把bitcount计数变量设为了函数的形参，一旦离开函数形参被自动释放，这就导致没有输出的比特位虽然已经被读取到但是没有被写入，因此我将写入的字符byte和比特计数变量bitcount都设为类成员，解决了编码时跳位写入的问题。

然而解码时，发现读到了正确的哈夫曼编码，程序不跳出识别编码，还要继续识别其他：

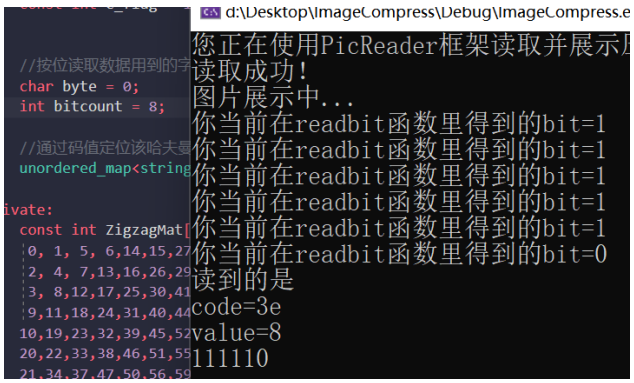


图 4.5-1 比特计数位初始化问题

因为初始我设置bitcount=8，那么第一个char在函数第一次执行的时候永远都读不到，从而导致解码始终错位，为了保证第一次就能进入if分支，我们需要将类成员bitcount的初始值设为0：

图 4.5-2 比特计数位初始值修改正确

4.6 哈希表的键值

起初想直接用struct Code的结构体作为哈希表的键值，但编译报错。于是想直接用Code结构体的成员code（int型）作键值，但是出现了如下问题：

图 4.6-1 哈希表的键值对应方式选取有误

哈希表对应方式不行：第一个是d0，对应的value应该是5，但是这样找读的话，读到11，就直接去查到011（value=2）了所以不可以。将哈希表的键值直接设为二进制码对应的string，增添一个二进制码转string的函数即可：

```
//将计算出的code值转换为string类
string Decode::code_to_str(int code,int code_len) {
    string str = "";
    int bit;
    //按位与的遮罩
    int mask[] = { 1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768 };
    for (int i = 0; i < code_len; i++) {
        bit = code & mask[code_len - i - 1];
        if (bit)
            str += '1';
        else
            str += '0';
    }
    return str;
}
```

图 4.6-2 int型code值转换为string

5 心得体会

Lena你害人不浅！从未写一个程序写的我如此崩溃，前前后后写了一周有余，而且因为开始太晚，最后差一点爆肝ddl。

总的来说这次作业还是受益匪浅的，不过lena带给我更多的是一条关于编程时心态的教训，这周二因为屡屡de不出bug而心态崩溃，高呼自己是所有共学oop课程的朋友里最愚蠢最不具资质的一位，几度想要放弃解码部分。然而事实就是如此，静下心来单步打出每一步的编码情况，不到一个小时问题也就迎刃而解了。仔细想想周二晚上面对满屏的乱码痛哭流涕的样子不免有些可爱，写程序果真最忌讳的就是心急如焚！后来的解码部分，谨记时刻心态平和，用了一个下午的时间也就较为轻松地完成了，教训如斯！

日常生活中使用各种软件压缩图片大小时有发生，这一次可以用自己写的小程序压缩图片，实现了较高的压缩率，内心也的确十分满足。JPEG格式的图片暗藏了t太多玄机，线性代数、傅里叶变换等等数学知识，让一个个小小的像素点焕发出不一般的活力。

本次作业采用类封装的形式实现，以在debug部分提到的bitcount问题为例，我们知道设为全局变量也可以解决问题，但程序的安全性就大大折扣。而我们将其设为类成员，在起到同样效果的同时保证了安全性，语义上也更符合逻辑，这让我浅尝到class的魅力。

最后还是对自己说一句：

认真debug，少犯蠢！

6 源代码

6.1 Compress.h

```
/*计算机 2252429 蔡宇轩*/
#pragma once
#include<iostream>
#include<fstream>
#include<iomanip>
#include<string>
using namespace std;

struct Code {
    int code;//Huffman 编码的值
    int length;//Huffman 编码的长度
};

class JPEG {
public:
    UINT img_width;
    UINT img_height;
    BYTE* data;//原始读入的 RGBA 数据
    char* Picname;//原始文件名
    ofstream outfile;//写入压缩文件的文件输入流
private:
    string outstr;
    int bitcount = 0;//当前已有的比特位数，每满 8 位输出 1 次
    char byte = 0;//向文件中写入的字节

    Code Y_dc[16] = { 0 };//Y-DC 哈夫曼编码表
    Code C_dc[16] = { 0 };//cbcr-DC 哈夫曼编码表
    Code Y_ac[256] = { 0 };//Y-AC 哈夫曼编码表
    Code C_ac[256] = { 0 };//cbcr-ACh 哈夫曼编码表
    Code* HuffTable[4] =
{ Y_dc,C_dc,Y_ac,C_ac };//4 个哈夫曼表的索引表
    Code* AC_table[2] = { Y_ac,C_ac };//AC 对应的哈夫曼表索引
    Code* DC_table[2] = { Y_dc,C_dc };//DC 对应的哈夫曼表索引
    const int Y_flag = 0;//preflag 可以共用
    const int C_flag = 1;//preflag 可以共用
    const int Cr_pre_flag = 2;
    int preDC[3] = { 0 };//preDC[0]-Y,preDC[1]-Cb,preDC[2]-Cr
    //标准量化系数表 Y
    const unsigned char
Luminance_Quantization_Table[64] =
    {
        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
    };
    //标准量化系数表 cbcr
    const unsigned char
Chrominance_Quantization_Table[64] =
    {
         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 int ZigzagMat[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
    };
    const int data_length = 64;
    //YCbCr 的 ACDC 表
    //Y-DC
    const unsigned char DC_Y_NRCodes[16] =
{ 0,1,5,1,1,1,1,1,1 };
    const unsigned char DC_Y_Values[16] =
{ 0,1,2,3,4,5,6,7,8,9,10,11 };
    //C-DC
    const unsigned char DC_C_NRCodes[16] =
{ 0,3,1,1,1,1,1,1,1,1,1 };
    const unsigned char DC_C_Values[16] =
{ 0,1,2,3,4,5,6,7,8,9,10,11 };
    //Y-AC
    const unsigned char AC_Y_NRCodes[16] =
{ 0,2,1,3,3,2,4,3,5,5,4,4,0,0,1,125 };
    const unsigned char AC_Y_Values[256] =
    {
        0x01,0x02,0x03,0x00,0x04,0x11,0x05,0x12
,0x21,0x31,0x41,0x06,0x13,0x51,0x61,0x07,0x22,0
x71,0x14,
        0x32,0x81,0x91,0xa1,0x08,0x23,0x42,0xb1
,0xc1,0x15,0x52,0xd1,0xf0,0x24,0x33,0x62,0x72,0
x82,0x09,0x0a,0x16,0x17,0x18,0x19,
        0x1a,0x25,0x26,0x27,0x28,0x29,0x2a,0x34
,0x35,0x36,0x37,0x38,0x39,0x3a,0x43,0x44,0x45,0
x46,0x47,0x48,0x49,0x4a,0x53,0x54,
        0x55,0x56,0x57,0x58,0x59,0x5a,0x63,0x64
,0x65,0x66,0x67,0x68,0x69,0x6a,0x73,0x74,0x75,0
x76,0x77,0x78,0x79,0x7a,0x83,0x84,
        0x85,0x86,0x87,0x88,0x89,0x8a,0x92,0x93
,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0
xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,
        0xb2,0xb3,0xb4,0xb5,0xb6,0xb7,0xb8,0xb9
,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0
xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,
        0xd8,0xd9,0xda,0xe1,0xe2,0xe3,0xe4,0xe5
,0xe6,0xe7,0xe8,0xe9,0xea,0xf1,0xf2,0xf3,0xf4,0
xf5,0xf6,0xf7,0xf8,0xf9,0xfa
    };
    //C-AC
    const unsigned char AC_C_NRCodes[16] =
{ 0,2,1,2,4,4,3,4,7,5,4,4,0,1,2,119 };
    const unsigned char AC_C_Values[256] =
    {
        0x00,0x01,0x02,0x03,0x11,0x04,0x05,0x21
,0x31,0x06,0x12,0x41,0x51,0x07,0x61,0x71,0x13,0
x22,0x32,0x81,0x08,0x14,0x42,0x91,
        0xa1,0xb1,0xc1,0x09,0x23,0x33,0x52,0xf0
,0x15,0x62,0x72,0xd1,0x0a,0x16,0x24,0x34,0xe1,0
x25,0xf1,0x17,0x18,0x19,0x1a,0x26,0x27,0x28,0x2
9,0x2a,0x35,0x36,
        0x37,0x38,0x39,0x3a,0x43,0x44,0x45,0x46
,0x47,0x48,0x49,0x4a,0x53,0x54,0x55,0x56,0x57,0
x58,0x59,0x5a,0x63,0x64,0x65,0x66,0x67,0x68,0x6
9,0x6a,0x73,0x74,
        0x75,0x76,0x77,0x78,0x79,0x7a,0x82,0x83
,0x84,0x85,0x86,0x87,0x88,0x89,0x8a,0x92,0x93,0
x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0xa
4,0xa5,0xa6,0xa7,
        0xa8,0xa9,0xaa,0xb2,0xb3,0xb4,0xb5,0xb6
,0xb7,0xb8,0xb9,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0
xc7,0xc8,0xc9,0xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd
7,0xd8,0xd9,0xda,
        0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9
,0xea,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,
0xfa
    };
    const unsigned char* ValuesPtr[4] =
{ DC_Y_Values,DC_C_Values,AC_Y_Values,AC_C_Valu
es };
    //得到一个数字 code 的第 length 位的值
    int GetBit(unsigned int code, int length);
    //写入一个字长的数据时需要高低位互换
    void write_word(int word,ofstream&
outfile);
    //打表，写入文件头
    void writeJPEGhead(ofstream& outfile);
    //打表，写入文件尾
    void writeJPEGtail(ofstream& outfile);
    //色彩通道转换，从 rgba 转换为 Ycbcr
    void cgto_YCbCr(BYTE* RGBbuffer,double
Ydata[],double Cbdata[],double Crdata[]);
    //傅里叶 DCT 变换、量化处理并 zigzag 到一维数组
里
    void cgto_DCT(double data[], int
dct_data[], const unsigned char QuantTable[]);
    //建立四张 ACDC 表的哈夫曼编码表
    void BuildHuffTable();
    //DC 编码过程
    void DC_bitcnt(int TABLEFLAG, int
dct_z_data, Code HuffCode[], int& index,int
PREFLAG);
    //AC 编码过程
    void AC_bitcnt(int TABLEFLAG, int eob, int
dct_z_data[], Code* HuffCode, int& index);
    //Huffman 编码
    int Huff_encoding(int flagtable,int
dct_data[],Code HuffCode[],int PREFLAG);
    //写入比特流文件
    void write_bits(Code HuffCode[], const int
index);
public:
    //构造函数
    JPEG(BYTE* readData,UINT readWidth,UINT
readHeight,char readName[]) {
        img_width = readWidth;
        img_height = readHeight;
        data = readData;
        Picname = readName;
    }
    //析构函数
    ~JPEG(){};
    //压缩函数
    void MyCompress();
};
```

6.2 Decode.h

```
/*计算机 2252429 蔡宇轩*/
#pragma once
#include"Compress.h"
#include<unordered_map>

class Decode {
    //图像的基本信息
    PicReader imread;
    BYTE* data;//最后要还原回去的三通道 RGB 数据
    UINT img_width;//图片宽度
    UINT img_height;//图片高度
    ifstream infile;//读取压缩图片文件的文件输入流
private:
    //操作过程中的中间量
    //4 个哈夫曼码的 AC DC 表
    unsigned char DC_Y_NRCodes[16] = { 0 };
    unsigned char DC_Y_Values[16] = { 0 };

    unsigned char DC_C_NRCodes[16] = { 0 };
    unsigned char DC_C_Values[16] = { 0 };

    unsigned char AC_Y_NRCodes[16] = { 0 };
    unsigned char AC_Y_Values[256] = { 0 };

    unsigned char AC_C_NRCodes[16] = { 0 };
    unsigned char AC_C_Values[256] = { 0 };

    unsigned char* NRCodes_table[4] =
{ DC_Y_NRCodes, DC_C_NRCodes, AC_Y_NRCodes,
AC_C_NRCodes };
    unsigned char* Values_table[4] =
{ DC_Y_Values, DC_C_Values, AC_Y_Values,
AC_C_Values };

    //量化表
    unsigned char
Luminance_Quantization_Table[64] = { 0 };
    unsigned char
Chrominance_Quantization_Table[64] = {0};

    //3 个 YCbCr 对应 DC 的上一个 DC 值
    int preDC[3] = { 0,0,0 };
    const int Y_flag = 0;
    const int C_flag = 1;

    //按位读取数据用到的字符型和比特计数量
    char byte = 0;
    int bitcount = 0;

    //通过码值定位该哈夫曼码在哈夫曼表中的位置-->哈希表存储
    unordered_map<string, int>
FindinHuffTable[4];//第一个 int 是 key（code），第二个 int 是 value（位序）

private:
    const int ZigzagMat[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
    };

private:
    //读取文件头
    bool readJPEGhead();
    //依据文件头的信息重建哈夫曼编码的 ACDC 表
    void readHuffTable(int tag);
    //重建哈夫曼编码表和对应的哈希表
    void BuildHuff_and_Hash();
    //读一个 8*8 矩阵对应的编码
    void readLine(int FLAG,int zdata[],int
preFlag);
    //反 zigzag、反 DCT
    void reverseDCT(int FLAG,int zdata[],
double YCbCrT[]);
    //转换回 GRB 三通道
    void cgto_RGB(double Y[], double Cb[],
double Cr[],int x,int y);
    //向下通过查询哈希表读取一个 Code
    int GetCode(int flag);
    //在文件中继续向下读取 1 个比特位
    int read_next_bit();
    //将从文件中读到的高低位倒序的数字正序计算出来并返回
    int cgto_num(short word);
    //将计算出的 code 值转换为 string 类
    string code_to_str(int code,int code_len);
public:
    //解压函数
    void MyDecode();
    //构造函数
    Decode() {};
    //析构函数
    ~Decode() {};
};
```

6.3 Image_tools.cpp

1.编码部分：

```
/*计算机 2252429 蔡宇轩*/
#include "PicReader.h"
#include "Compress.h"
#include "Decode.h"


//求一个 8*8 像素矩阵对应的 ycbcr 矩阵
void JPEG::cgto_YCbCr(BYTE* RGBbuffer, double
Ydata[], double Cbdata[], double Crdata[]) {
    //index 是相对于 RGBbuffer 的偏移量
    int index = 0;
    for (int i = 0; i < 8; i++) {
        int first = index;
        for (int j = 0; j < 8; j++) {
            double R =
(double)RGBbuffer[index], G =
(double)RGBbuffer[index + 1], B =
(double)RGBbuffer[index + 2];
            Ydata[i * 8 + j] = 0.29871 * R +
0.58661 * G + 0.11448 * B - 128;
            Cbdata[i * 8 + j] = -0.16874 * R -
0.33126 * G + 0.5 * B;
```

```
            Crdata[i * 8 + j] = 0.5 * R -
0.41869 * G - 0.08131 * B;
            index += 4;
        }
        index = first + img_width * 4;
    }
}
/**********将一个 8 * 8 矩阵 Mat 作傅里叶变换并量化
***********/
//data[]原始数据
        //dct_data 经过 DCT 变换和 zigzag 处理后的
一维数组
        //QuantTable 量化表，Y 和 CbCr 有不同的量化
表，常量
void JPEG:: cgto_DCT(double data[], int
dct_data[], const unsigned char QuantTable[]) {
    //DCT 变换的本质是 data[]经过基变换转换到
dct_data[]
    //dct_data[]是以 64 个标准图像为基时对应的系数
    const double pi = 3.1415926;//定义 pai 的取值
    for (int u = 0; u < 8; u++) {
        for (int v = 0; v < 8; v++) {
            //本质是二维矩阵，2 个 alpha 系数
            double alpha_u = (u == 0) ? 1.f /
sqrt(8.f) : 0.5f;
            double alpha_v = (v == 0) ? 1.f /
sqrt(8.f) : 0.5f;

            //系数矩阵未量化前的值记录在临时变量
tmp 中
            double tmp = 0;
            //此处作双层的累加
            for (int x = 0; x < 8; x++) {
                for (int y = 0; y < 8; y++) {
                    //当前 data 中的对应的矩阵元素
值
                    double cur_data = data[x *
8 + y];
                    //求 DCT 基变换系数的公式
                    cur_data *= cos((2 * x + 1)
* u * pi / 16.f) * cos((2 * y + 1) * v * pi /
16.f);
                    tmp += cur_data;//做累加
                }
            }
            tmp *= alpha_u * alpha_v;
            ////tmp 的值量化取整
            int tmp_c = (int)round(tmp /
(double)QuantTable[u * 8 + v]);//除以量化表对应
位置的系数即可
            ////直接 zigzag 存入 dct_data[]
            short ZigzagIndex = ZigzagMat[u * 8
+ v];
            dct_data[ZigzagIndex] = tmp_c;
        }
    }
}

//编码 Huffman
void JPEG::BuildHuffTable() {
    const unsigned char* every_code_len[4] = //
每个编码表的不同码长下对应的码数量
    { DC_Y_NRCodes,DC_C_NRCodes,AC_Y_NRCodes,AC
_C_NRCodes };
    const unsigned char* start[4] =//每个编码表
的不同码长对应的一组码的开始位置
    { DC_Y_Values,DC_C_Values,AC_Y_Values,AC_C_
Values };
    //type 表示当前创建的编码表种类
    //HuffTable 是装有 4 个编码表索引的大表
    for (int type = 0; type < 4; type++) {
        int index = 0;//在自己的表中对应的下标索
引
        int code = 0;//编码
        Code* p = HuffTable[type];
        for (int i = 0; i < 16; i++) {
            //cout << "now length=" << i + 1 <<
endl;
            for (int j = 0; j <
*(every_code_len[type] + i); j++) {
                //没有长度为 0 的编码，码长从 1 开
始
                //范式哈夫曼：同一长度的码长相邻只
差 1
                p[ValuesPtr[type][index]].code
= code;
                p[ValuesPtr[type][index]].lengt
h = i + 1;
                code++;
                index++;
            }
            code <<= 1;//相邻码长：某长度开始的第
一个编码等于上一个长度的最后一个编码左移一位
        }
    }
}

/***************************bit 表部分
*********************************/
//完成游程 RLE 编码和相应 AC、DC 部分的数值转化
//int eob 表示 EOB 在 data 中的位置
//AC 编码
void JPEG::AC_bitcnt(int TABLEFLAG, int eob,
int dct_z_data[], Code* HuffCode, int& index) {
    //判断是否是（15,0）的标志位,15*16+0=0xf0=240
    bool FZR = false;
    Code combine_FZR= *(AC_table[TABLEFLAG] +
0xf0);//huffcode for (15,0)FZR

    //记录连续 0 的数量
    int zerocnt = 0;
    int i = 1;
    //我们只能操作到数组下标 eob 以前
    while(i<eob){

        //当前读到的值是 dct_z_data[i]

        //记录连续 0
        while (dct_z_data[i] == 0) {
            //cout << i << endl;
            zerocnt++;
            if (i >= eob) {
                i--;
                zerocnt--;
                break;
            }
            //若有连续的 16 个 0 则需跳出
            if (zerocnt >= 16) {
                FZR = true;
```

```
                break;
            }//判断 FZR
            i++;
        }// counting num of zero
        if (FZR) {
            //只存储 FZR 即可，无需存储 bit 码
            HuffCode[index] = combine_FZR;
            index++;
            zerocnt = 0;//计数量清零
            FZR = false;//标志位清零
        }//FZR
        else {
            //dct_z_data 的绝对值
            int abs_val =
(int)fabs(dct_z_data[i]);
            Code BitCode = { 0 };
            BitCode.length = 0; BitCode.code =
0;
            //计算 abs_val 的二进制数长度
            while (abs_val > 0) {
                abs_val >>= 1;
                BitCode.length++;
            }//length of abs_val=length of
bitcode
            abs_val =
(int)fabs(dct_z_data[i]);//还原 abs_val

            //首先编哈夫曼码
            unsigned char combine = zerocnt *
16 + BitCode.length;
            HuffCode[index] =
*(AC_table[TABLEFLAG] + combine);
            index++;
            //比特码
            if (dct_z_data[i] > 0) {//正数
                //BitCode 的值是其本身
                BitCode.code = dct_z_data[i];
                HuffCode[index] = BitCode;
                index++;
                zerocnt = 0;
            }
            else if (abs_val != dct_z_data[i])
{//负数
                BitCode.code = (int)pow(2,
BitCode.length) - 1 +dct_z_data[i];
                HuffCode[index] = BitCode;
                index++;
                zerocnt = 0;
            }
        }
        i++;
    }
}
//DC 编码
void JPEG::DC_bitcnt(int TABLEFLAG, int
dct_z_data, Code HuffCode[], int& index, int
PREFLAG) {

    //计算本次的 DC 与上一个 DC 的差值
    int difference = dct_z_data -
preDC[PREFLAG];
    preDC[PREFLAG] = dct_z_data;//更新 pre

    //difference 的绝对值
    int abs_val = (int)fabs(difference);
    //bit 码
    Code BitCode = { 0 };
    //计算 abs_val 的二进制数长度
    while (abs_val > 0) {
        abs_val >>= 1;
        BitCode.length++;
    }//length of abs_val=length of bitcode

    //第一位的合并字符,哈夫曼码
    unsigned char combine = 16 * 0 +
BitCode.length;


    //difference 与 0 合并成为的 combine 作为第一个
哈夫曼码
    HuffCode[index] = *(DC_table[TABLEFLAG] +
combine);
    index++;//Huffman

    if (difference != 0) {//第二位存储比特码（只
有 difference=非零数才存）
        abs_val = (int)fabs(difference);
        if (difference > 0) {//正数
            //BitCode 的值是其本身
            BitCode.code = difference;
        }
        else if (abs_val != difference) {//负数
            BitCode.code = (int)pow(2,
BitCode.length) - 1 + difference;
        }
        HuffCode[index] = BitCode;
        index++;
    }
}


//TABLEFLAG:标记当前是 Y 还是 C 表
//dct_data 是已经量化并 zigzag 的数据
int JPEG::Huff_encoding(int TABLEFLAG,int
dct_z_data[],Code HuffCode[],int PREFLAG) {
    int index = 0;//当前码的下标索引
    /*****************************DC 编码
*****************************/
    DC_bitcnt(TABLEFLAG, dct_z_data[0],
HuffCode, index, PREFLAG);
    /*****************************DC 编码
*****************************/

    /*****************************AC 编码
*****************************/
    //找到 dct_data 中 EOB 的位置
    int i = 63;
    while (dct_z_data[i] == 0)
        i--;
    int eob = ++i;//找到末尾第一个 0 出现的位置
    //编码 eob 之前的 AC 部分
    if (eob > 1) {
        AC_bitcnt(TABLEFLAG, eob, dct_z_data,
HuffCode, index);
    }
    //写入 EOB
    Code EOB = AC_table[TABLEFLAG][0x00];//EOB-
>0
    if (eob <= 63) {
        HuffCode[index] = EOB;
        index++;
    }
```

```
    /*****************************AC 编码
*******************************/
    return index;//返回整个编码的长度
}
void JPEG:: write_bits(Code HuffCode[],const
int index) {
    int lala = 0;
    //cout << "writing..." << endl;
    for (int i = 0; i < index; i++) {
        int code = HuffCode[i].code;
        int length = HuffCode[i].length;
        while (length>0) {
            //每满 8 位输出 1 个字节
            unsigned char bit = (unsigned
char)GetBit(code, length);//当前位数下 code 的比
特位值
            byte <<= 1;//待输出字符左移 1 位
            if (bit)
                byte |= 1;
            bitcount++;
            length--;//已经输出则长度-1，当
length 减为 0 时一次循环结束，读数组下一个编码
            if (bitcount == 8) {
                outstr += byte;
                //与 JPEG 的 FF 标记区分
                if (byte == (char)0xff) {
                    outstr += '\0';
                }
                //清空
                byte = 0;
                bitcount = 0;
            }
        }
    }
}
void JPEG:: MyCompress() {
    //创建并打开写入文件
    outfile.open("lena.jpg", ios::binary);
    if (!outfile) {
        cerr << "failed to open output file!"
<< endl;
        exit(-1);
    }
    //写入文件头
    writeJPEGhead(outfile);
    //生成哈夫曼表
    BuildHuffTable();
    //开始编码
    //每 8*8 个矩阵快操作一次
    for (unsigned int x = 0; x < img_height; x
+= 8) {
        for (unsigned int y = 0; y < img_width;
y += 8) {
            /*cout << "现在是第"<<times << "个
8*8 矩阵：" << endl;*/
            //当前 8*8 矩阵左上角像素点所在的数组下
标
            BYTE* RGBbuffer = data + x *
img_width * 4 + y * 4;//每个像素点都有 rgba 四个值
            //由 RGB 转化为 ycbcr 三个矩阵
            double Y_data[64] = { 0 },
Cb_data[64] = { 0 }, Cr_data[64] = { 0 };
            cgto_YCbCr(RGBbuffer, Y_data,
Cb_data, Cr_data);
            //DCT
            int Y_dct[64] = { 0 }, Cb_dct[64] =
{ 0 }, Cr_dct[64] = { 0 };
            int index=0;
            //压缩 Y 通道
            cgto_DCT(Y_data, Y_dct,
Luminance_Quantization_Table);
            Code* HuffCode_Y =
new(nothrow)Code[128];//最多有 64 个，每一个都分成
2 部分
            if (!HuffCode_Y) {
                cerr << "failed to allocate!"
<< endl;
                exit(-1);
            }
            index=Huff_encoding(Y_flag, Y_dct,
HuffCode_Y,Y_flag);//index????????????
            write_bits(HuffCode_Y,index);
            delete[] HuffCode_Y;

            //压缩 Cb 通道
            cgto_DCT(Cb_data, Cb_dct,
Chrominance_Quantization_Table);
            Code* HuffCode_Cb =
new(nothrow)Code[128];
            if (!HuffCode_Cb) {
                cerr << "failed to allocate!"
<< endl;
                exit(-1);
            }
            index=Huff_encoding(C_flag, Cb_dct,
HuffCode_Cb,C_flag);
            write_bits(HuffCode_Cb,index);
            delete[] HuffCode_Cb;


            //压缩 Cr 通道
            cgto_DCT(Cr_data, Cr_dct,
Chrominance_Quantization_Table);
            Code* HuffCode_Cr =
new(nothrow)Code[128];
            if (!HuffCode_Cr) {
                cerr << "failed to allocate!"
<< endl;
                exit(-1);
            }
            index=Huff_encoding(C_flag, Cr_dct,
HuffCode_Cr,Cr_pre_flag);
            write_bits(HuffCode_Cr,index);
            delete[] HuffCode_Cr;
        }
    }
    writeJPEGtail(outfile);
    outfile.close();
    return;
}
int JPEG::GetBit(unsigned int code, int length)
{

    int bit;
    //按位与的遮罩
    int mask[] =
{ 1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8
192,16384,32768 };
    bit = code & mask[length - 1];
    if (bit)
        return 1;
    else
```

```
        return 0;
}
void JPEG::write_word(int
word,ofstream&outfile) {
    char low = (char)(word & 0xff);
    char high = (char)(word >> 8);
    outfile.write((char*)&high, sizeof(char));
    outfile.write((char*)&low, sizeof(char));
}
//写文件头
void JPEG::writeJPEGhead(ofstream& outfile){
    //SOI
    write_word(0xffd8, outfile);
    //APP0
    write_word(0xffe0, outfile);
    write_word(16, outfile);//数据长度 16
    //标识符"JFIF"
    char JFIF_str[5] = "JFIF";
    outfile.write((char*)JFIF_str, 5);

    //版本
    unsigned char version = 1;
    outfile.write((char*)&version,
sizeof(version));
    //版本
    outfile.write((char*)&version,
sizeof(version));

    outfile << (unsigned char)0;//X 和 Y 的密度单
位
    write_word(1, outfile);//X 方向像素密度
    write_word(1, outfile);//Y 方向像素密度
    write_word(0, outfile);//缩略图水平和垂直像素
数目，无缩略图


    //DQT
    write_word(0xffdb, outfile);//标识符
    write_word(132, outfile);//数据长度
    //精度和类型
    outfile.put(0);
    //zigzag 写入第一个量化表-Y
    unsigned char Qtable[64] = { 0 };
    for (int i = 0; i < 64; i++)
        Qtable[ZigzagMat[i]] =
Luminance_Quantization_Table[i];
    outfile.write((char*)Qtable, 64);


    outfile.put(1);
    //zigzag 写入第二个量化表 CbCr
    for (int i = 0; i < 64; i++)
        Qtable[ZigzagMat[i]] =
Chrominance_Quantization_Table[i];
    outfile.write((char*)Qtable, 64);

    //SOF0
    write_word(0xffc0, outfile);//标记符
    write_word(17, outfile);//数据长度
    //图像精度
    outfile.put(8);
    write_word(img_height & 0xffff, outfile);//
图像高度
    write_word(img_width & 0xffff, outfile);//
图像宽度


    //颜色分量数恒为 3
    outfile.put(3);
    //3 个颜色分量 YCbCr
    //Y
    outfile.put(1);//颜色分量 ID-1-Y 标志
    outfile.put(0x11);//水平和垂直采样因子
    outfile.put(0);//当前分量使用的量化表 ID
YTable id=0
    //Cb
    outfile.put(2);//颜色分量 ID-2-Cb
    outfile.put(0x11);//水平和垂直采样因子
    outfile.put(1);//当前分量使用的量化表 ID
CTable id=1

    //Cr
    outfile.put(3);//颜色分量 ID-2-Cb
    outfile.put(0x11);//水平和垂直采样因子
    outfile.put(1);//当前分量使用的量化表 ID
CTable id=1


    //DHT
    write_word(0xffc4, outfile);//标识符
    //计算 DHT 段数据长度
    int len[4] = { 0 };
    for (int i = 0; i < 16; i++) {
        len[0] += DC_Y_NRCodes[i];
        len[1] += DC_C_NRCodes[i];
        len[2] += AC_Y_NRCodes[i];
        len[3] += AC_C_NRCodes[i];
    }
    int length = len[0] + len[1] + len[2] +
len[3] + 2 + 4 + 4 * 16;
    //数据长度计算完毕
    write_word(length, outfile);//写入数据长度

    //给 Y 和 CbCr 分别编 DC,AC 表
    //HT-Y-DC
    outfile.put(0x00);
    outfile.write((char*)DC_Y_NRCodes, 16);
    outfile.write((char*)DC_Y_Values, len[0]);


    //HT-Y-AC
    outfile.put(0x10);
    outfile.write((char*)AC_Y_NRCodes, 16);
    outfile.write((char*)AC_Y_Values, len[2]);


    //HT-CbCr-DC
    outfile.put(0x01);
    outfile.write((char*)DC_C_NRCodes, 16);
    outfile.write((char*)DC_C_Values, len[1]);


    //HT-CbCr-AC
    outfile.put(0x11);
    outfile.write((char*)AC_C_NRCodes, 16);
    outfile.write((char*)AC_C_Values, len[3]);


    //SOS
    write_word(0xffda, outfile);//标识符
```

```
    write_word(12, outfile);//数据长度
    //颜色分量数 Ycbcr=3
    outfile.put(3);

    //颜色分量信息
    outfile.put(1); outfile.put(0x00);//y
    outfile.put(2); outfile.put(0x11);//cb
    outfile.put(3); outfile.put(0x11);//cr

    //压缩图像数据
    outfile.put(0x00);//谱选择开始
    outfile.put(0x3f);//谱选择结束
    outfile.put(0x00);//谱选择

}
//写文件尾
void JPEG::writeJPEGtail(ofstream& outfile) {
    if (bitcount > 0 && bitcount < 8) {
        byte <<= (8 - bitcount);
        outstr += byte;
    }
    outfile << outstr;
    //EOI
    write_word(0xffd9, outfile);
}
```

2.解码部分：

```
/***************解码部分****************/
//读取文件头存储 4 个哈夫曼 AC DC 表以及图片的基本信息，返回值为是否读取成功
bool Decode::readJPEGhead() {
    infile.open("lena.jpg", ios::binary);
    if (!infile) {
        cerr << "failed to open compressed file!" << endl;
        return false;
    }
    //只需要图片的基本信息，JPEG 格式的文件头实则不需要
    //这里定义不同长度的缓冲变，方便读取
    unsigned char byte=0;//1 字节
    unsigned short word=0;//字=2 字节
    unsigned int dword=0;//双字=4 四节

    //读取 SOI
    infile.read((char*)&word, sizeof(word));//0xffd8
    //读取 APPO
    infile.read((char*)&dword, sizeof(dword));//0xffe0,0x16
    //读取标识符 JFIF
    char JFIF[5] = { 0 };
    infile.read(JFIF, 5);
    //版本
    infile.read((char*)&byte, sizeof(byte));//version1
    infile.read((char*)&byte, sizeof(byte));//version2
    //X,Y 密度单位
    infile.read((char*)&byte, sizeof(byte));
    infile.read((char*)&word, sizeof(word));//X 方向像素密度
    infile.read((char*)&word, sizeof(word));//Y 方向像素密度
    infile.read((char*)&word, sizeof(word));//缩略图水平和垂直像素数目，无缩略图

    //读取 DQT
    infile.read((char*)&word, sizeof(word));//标识符
    infile.read((char*)&word, sizeof(word));//数据长度

    //精度和类型
    infile.read((char*)&byte, sizeof(byte));//0
    unsigned char Qtable_z[64] = { 0 };//暂存读取到的 zigzag 之后的量化表
    //反 zigzag 还原 Y 量化表
    for (int i = 0; i < 64; i++) {
        infile.read((char*)&byte, sizeof(byte));
        Qtable_z[i] = (unsigned char)byte;
    }
    //反 zigzag 赋值给 Y 量化表
    for (int i = 0; i < 64; i++)
        Luminance_Quantization_Table[i] = Qtable_z[ZigzagMat[i]];


    infile.read((char*)&byte, sizeof(byte));//1
    //反 zigzag 还原 C 量化表
    for (int i = 0; i < 64; i++) {
        infile.read((char*)&byte, sizeof(byte));
        Qtable_z[i] = (unsigned char)byte;
    }
    //反 zigzag 赋值给 C 量化表
    for (int i = 0; i < 64; i++)
        Chrominance_Quantization_Table[i] = Qtable_z[ZigzagMat[i]];

    //读取 SOFO
    infile.read((char*)&word, sizeof(word));//0xffc0
    infile.read((char*)&word, sizeof(word));//length of data=17
    //图像精度
    infile.read((char*)&byte, sizeof(byte));//8
    infile.read((char*)&word, sizeof(word));//图像高度
    img_height = (UINT)cgto_num(word);

    infile.read((char*)&word, sizeof(word));//图像宽度
    img_width = (UINT)cgto_num(word);


    //颜色分量 YCbCr
    infile.read((char*)&byte, sizeof(byte));//颜色分量数=3
    //Y
    infile.read((char*)&byte, sizeof(byte)); //颜色分量 ID-1-Y 标志
    infile.read((char*)&byte, sizeof(byte));//水平和垂直采样因子 0x11
    infile.read((char*)&byte, sizeof(byte));//当前分量使用的量化表 ID YTable id=0
```

```
    //Cb
    infile.read((char*)&byte, sizeof(byte)); //颜色分量 ID-2-Cb 标志
    infile.read((char*)&byte, sizeof(byte));//水平和垂直采样因子 0x11
    infile.read((char*)&byte, sizeof(byte));//当前分量使用的量化表 ID CTable id=1
    //Cr
    infile.read((char*)&byte, sizeof(byte)); //颜色分量 ID-2-Cr 标志
    infile.read((char*)&byte, sizeof(byte));//水平和垂直采样因子 0x11
    infile.read((char*)&byte, sizeof(byte));//当前分量使用的量化表 ID CTable id=1

    //读取 DHT
    infile.read((char*)&word, sizeof(word));//标识符 0xffc4
    infile.read((char*)&word, sizeof(word));//数据长度

    //读取 4 个哈夫曼 ACDC 表
    for (int i = 0; i < 4; i++) {
        //当前表的标识记号 type=0~3
        //分别对应:0-y_dc,1-c_dc,2-y_ac,3-c_ac
        infile.read((char*)&byte, sizeof(byte));
        unsigned char tag = (unsigned char)byte;
        readHuffTable(tag);//生成 ACDC 表
    }
    //读取 SOS
    infile.read((char*)&word, sizeof(word));//标识符 0xffda
    infile.read((char*)&word, sizeof(word));//数据长度 12

    infile.read((char*)&byte, sizeof(byte));//颜色分量数 ycbcr=3

    //颜色分量信息
    infile.read((char*)&word, sizeof(word));//y
    infile.read((char*)&word, sizeof(word));//cb
    infile.read((char*)&word, sizeof(word));//cr

    //压缩图像数据
    infile.read((char*)&byte, sizeof(byte));//谱选择开始
    infile.read((char*)&byte, sizeof(byte));//谱选择结束
    infile.read((char*)&byte, sizeof(byte));//请选择

    return true;
}
void Decode::readHuffTable(int tag) {
    int len = 0;
    unsigned char nrc_codes[16] = { 0 };
    infile.read((char*)nrc_codes, 16);
    int type = 2 * (tag / 16) + tag % 16;

    for (int i = 0; i < 16; i++) {
        len += nrc_codes[i];//求 values 数组的数量
        *(NRCodes_table[type] + i) = nrc_codes[i];//存储 NRCodes 数组
    }
    //存储 values 数组
    for (int i = 0; i < len; i++) {
        infile.read((char*)&byte, sizeof(byte));
        *(Values_table[type]+i)= (unsigned char)byte;
    }
}

void Decode::BuildHuff_and_Hash() {
    const unsigned char* every_code_len[4] = //每个编码表的不同码长下对应的码数量
    { DC_Y_NRCodes,DC_C_NRCodes,AC_Y_NRCodes,AC_C_NRCodes };
    const unsigned char* start[4] =//每个编码表的不同码长对应的一组码的开始位置
    { DC_Y_Values,DC_C_Values,AC_Y_Values,AC_C_Values };
    //type 表示当前创建的编码表种类
    //HuffTable 是装有 4 个编码表索引的大表
    for (int type = 0; type < 4; type++) {
        int index = 0;//在自己的表中对应的下标索引
        int code = 0;//编码
        //Code* p = HuffTable[type];
        for (int i = 0; i < 16; i++) {
            //cout << "now length=" << i + 1 << endl;
            for (int j = 0; j < *(every_code_len[type] + i); j++) {
                //没有长度为 0 的编码，码长从 1 开始
                //范式哈夫曼：同一长度的码长相邻只差 1
                int value = (int)Values_table[type][index];
                string codestr = "";
                codestr = code_to_str(code, i + 1);
                //插入对应的哈希表
                FindinHuffTable[type].insert({codestr,value});

                code++;
                index++;
            }
            code <<= 1;//相邻码长：某长度开始的第一个编码等于上一个长度的最后一个编码左移一位
        }
    }
}

void Decode::MyDecode() {

    //读文件头
    if (readJPEGhead() == false) {
        cerr << "Failed to read! " << endl;
        return;
    }
    else
        cout << "Succeed to read! " << endl;
```

```
    BYTE* RGBdata =
new(nothrow)BYTE[(int)img_width *
(int)img_height * 4];
    if (!RGBdata) {
        cerr << "failed to allocate!" << endl;
        exit(-1);
    }
    data = RGBdata;

    cout << "Picture showing..." << endl;

    //重建赫夫曼树与哈希表
    BuildHuff_and_Hash();

    for (int x = 0; x < (int)img_height; x +=
8) {
        for (int y = 0; y < (int)img_width; y
+= 8) {
            //还原出的 zigzag 完成后的待编码数组
            int Y_zdata[64] = { 0 },
Cb_zdata[64] = { 0 }, Cr_zdata[64] = { 0 };
            //还原出的 zigzag 以前的、DCT 傅里叶变
换以前的数组
            double Y[64] = { 0 }, Cb[64] =
{ 0 }, Cr[64] = { 0 };
            //一次解码一行，即一个 8*8 矩阵
            //Y
            readLine(Y_flag, Y_zdata, 0);
            reverseDCT(Y_flag, Y_zdata, Y);
            //Cb
            readLine(C_flag, Cb_zdata, 1);
            reverseDCT(C_flag, Cb_zdata, Cb);
            //Cr
            readLine(C_flag, Cr_zdata, 2);
            reverseDCT(C_flag, Cr_zdata, Cr);

            //转换回 RGB 三通道数组
            cgto_RGB(Y, Cb, Cr, x, y);
        }
    }


    imread.showPic(data, img_width,
img_height);
    return;
}

//FLAG:0=y-dc,1=c-dc,2-y-ac,3=c-ac
void Decode::readLine(int FLAG,int zdata[],int
preFlag) {
    int index = 0;//zdata 数组的下标索引

    /***************DC 解码****************/
    int combine = GetCode(FLAG);

    int zerocnt = combine / 16;//连续 0 的数量
    int bit_len = combine % 16;//比特码的长度
    int value = 0;//哈夫曼编码对应的真正的原值
    while (zerocnt > 0) {
        zdata[index++] = 0;
        zerocnt--;
    }
    for (int i = 0; i < bit_len; i++) {
        value *= 2;
        value += read_next_bit();
    }//get real value in zdata[]
    //判断正负数：正数的高位为 1，所以一定大于
2^(len-1)
    if (value >= pow(2, bit_len - 1))//positive
        preDC[preFlag] += value;
    else
        preDC[preFlag] += value - ((int)pow(2,
bit_len) - 1);
    //DC 存储编码的时候用的是 2 次差值，还原
    zdata[index++] = preDC[preFlag];
    /***************DC 解码****************/

    /***************AC 解码****************/
    //两个特殊的其后没有 bit 码的哈夫曼编码
    int combine_EOB = 0, combine_FZR = 15 * 16
+ 0;
    while (index < 64) {
        combine = GetCode(FLAG+2);
        if (combine == combine_EOB) {
            while (index < 64) {
                zdata[index++] = 0;
            }
            break;
        }//读到 EOB 直接填满 0，然后离开循环结束函数
就可以
        else if (combine == combine_FZR) {
            int fill_zero = 0;
            while (fill_zero < 16) {
                zdata[index++] = 0;
                fill_zero++;
            }
        }//读到 FZR=(15,0)直接连续填 16 个 0 即可，
        else {
            //普通的 AC 编码，其解码过程和 DC 解码是
完全相同的
            zerocnt = combine / 16;
            bit_len = combine % 16;
            value = 0;
            while (zerocnt > 0) {
                zdata[index++] = 0;
                zerocnt--;
            }//add zero
            for (int i = 0; i < bit_len; i++) {
                value <<= 1;
                value += read_next_bit();
            }//get real value in zdata[]
            //判断正负数：正数的高位为 1，所以一定
大于 2^(len-1)
            if (value >= (int)pow(2, bit_len -
1))
                zdata[index++] = value;
            else
                zdata[index++] = value -
((int)pow(2, bit_len) - 1);
        }
    }
}

//在文件中读取到一个编码并返回该编码对应的实际值
int Decode::GetCode(int FLAG) {
    bool toRead = true;
    int length = 0;
    int code = 0;
    int value = 0;//返回值：查询到的码所对应的实际
值
    string tosearch_code = "";
    while (toRead) {
```

```
        int bit= read_next_bit();
        code <<= 1;
        code += bit;//读取文件中的下一个比特位
        length++;
        tosearch_code= code_to_str(code,
length);

        //在哈希中寻找这个键值
        auto myCode =
FindinHuffTable[FLAG].find(tosearch_code);
        if (myCode !=
FindinHuffTable[FLAG].end()) {
            toRead = false;
            value = myCode->second;
        }//if found
    }
    return value;
}

//在文件中向下读取 1 个比特位
int Decode::read_next_bit() {
    //如果上一次读取到的 byte 还没用完则不需要进入这
个分支
    if (bitcount == 0) {//byte 中已经没有余量，继
续从文件中读
        infile.read((char*)&byte,
sizeof(byte));
        bitcount = 8;//重置可以数的的比特位
        if (byte == (char)0xff) {
            char zero = 0;
            infile.read((char*)&zero,
sizeof(zero));
            //FF 后面跟着的 00 需要直接吞掉
        }
    }
    int bit = byte & 128;
    byte <<= 1;//每次都是按位与比较 byte 的最高
位，为当前在文件中读取到的比特位
    bitcount--;//读完 1 个，比特计数量-1
    if (bit)
        return 1;
    else
        return 0;
}

void Decode::reverseDCT(int FLAG,int zdata[],
double YCbCr[]) {
    double before_zigzag[64] = { 0 };
    //首先反 zigzag
    for (int i = 0; i < 64; i++)
        before_zigzag[i] = zdata[ZigzagMat[i]];

    //反量化
    unsigned char* QTable = NULL;
    if (FLAG == Y_flag)
        QTable = Luminance_Quantization_Table;
    else
        QTable =
Chrominance_Quantization_Table;
    for (int i = 0; i < 64; i++)
        before_zigzag[i] *= QTable[i];

    //反 DCT
    const double pi = 3.1415926;
    for (int x = 0; x < 8; x++) {
        for (int y = 0; y < 8; y++) {
            //每一项暂记为 tmp
            double tmp = 0;
            //此处作双层的累加
            for (int u = 0; u < 8; u++) {
                for (int v = 0; v < 8; v++) {
                    //两个 C 系数
                    double c_u = (u == 0) ? 1.f
/ sqrt(8.f) : 0.5f;;
                    double c_v = (v == 0) ? 1.f
/ sqrt(8.f) : 0.5f;;
                    //当前 before_zigzag 数组中对
应的值
                    double cur_data =
before_zigzag[u * 8 + v];
                    //求 DCT 反变换的基变换系数公
式
                    cur_data *= cos((x + 0.5) *
pi * u / 8) * cos((y + 0.5) * pi * v / 8) * c_u
* c_v;
                    tmp += cur_data;//做累加
                }
            }
            YCbCr[x * 8 + y] = tmp;
        }
    }
}

//将 YCbCr 数组转化为 RGB
void Decode::cgto_RGB(double Y[], double Cb[],
double Cr[],int x,int y) {
    //(x,y)是矩阵左上角第一个点的像素位置
    int first = x * img_width + y;//first 是第一
个点在所有像素中的位序
    int index = 0;;
    int i = 0;
    //将 1 个 8*8 矩阵对应的 ycbcr 值还原回 rgb
    for (int x = 0; x < 8; x++) {//row
        for (int y = 0; y < 8; y++) {//col
            //在 data 中的索引
            index = (first + x * img_width +
y)*4;
            i = x * 8 + y;//在 ycbcr 矩阵中的索引
            int R =
(int)round(1.000200*(Y[i]+128) - 0.000992 *
Cb[i]+1.402126*Cr[i]);
            int G =
(int)round(1.000200*(Y[i]+128) - 0.345118 *
Cb[i] - 0.714010 * Cr[i]);
            int B =
(int)round(1.000200*(Y[i]+128) + 1.771018 *
Cb[i]+0.000143*Cr[i]);
            //处理 RGB 计算值的上下界溢出
            if (R < 0)R = 0;
            if (G < 0)G = 0;
            if (B < 0)B = 0;
            if (R > 255)R = 255;
            if (G > 255)G = 255;
            if (B > 255)B = 255;

            //cout << "index=" << index <<
endl;
            //cout << "R=" << R << " G=" << G
<< " B=" << B << endl;

            data[index] = (BYTE)R;
            data[index+1] = (BYTE)G;
```

```
            data[index+2] = (BYTE)B;
            ////RGBA 的 A
            data[index+3] = 255;
        }
    }
}

//将从文件中读到的高低位倒序的数字正序计算出来并返回
int Decode::cgto_num(short word) {
    int num_low= word >> 8;
    int num_high = (int)((char)word);
    int num = (num_high << 8) + num_low;
    return num;
}
//将计算出的 code 值转换为 string 类
string Decode::code_to_str(int code,int code_len) {
    string str = "";
    int bit;
    //按位与的遮罩
    int mask[] = { 1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768 };
    for (int i = 0; i < code_len; i++) {
        bit = code & mask[code_len - i - 1];
        if (bit)
            str += '1';
        else
            str += '0';
    }
    return str;
}
```

```
        cout << "Compressing..." << endl;
        JPEG my_jpeg(data, img_width, img_height,argv[2]);//构造并初始化一个 JPEG 类
        my_jpeg.MyCompress();//调用该类的 compress 函数
        cout << "[notice]:compressed filename \"lena.jpg\"" << endl;
        cout << "Complete!" << endl;
    }
    else if (!strcmp(argv[1], "-read")) {
        cout << "Reading..." << endl;
        cout << "[notice]:You are using [PicReader] to show picture [lena.jpg]!" << endl;
        Decode my_decode;
        my_decode.MyDecode();
        cout << "Complete!" << endl;
    }
    return 0;
}
```

6.4 Image_main.cpp

```
/*计算机 2252429 蔡宇轩*/
#include "PicReader.h"
#include "Compress.h"
#include "Decode.h"

int main(int argc,char**argv) {
    cout << "Zipper 0.002! Author:2252429" << endl;
    cout << "cmdline:[-read][-compress]" << endl;//输出的个性化提示
    if (argc != 3) {//参数个数错误
        cerr << "Please make sure the number of parameters is correct!" << endl;
        return -1;
    }
    if (strcmp(argv[1], "-read") && strcmp(argv[1], "-compress")) {//参数内容输入错误
        cerr<<"Invalid parameter!"<<endl;
        return -1;
    }
    if (!strcmp(argv[1], "-compress") ){
        /*********原图读取**********/
        PicReader imread;
        BYTE* data = nullptr;
        UINT img_width, img_height;
        imread.readPic(argv[2]);
        imread.getData(data, img_width, img_height);//每个像素点的 RGBA 存储在 data
        /*********原图读取**********/
```