《图像压缩》大作业报告

班级：智能交通与车辆 14 班

学号：2152402

姓名：段婷婷

完成日期：2022.5.5

目录

一. 功能描述与设计思路

1. 功能描述:

使用 **jpg 图像压缩**的方法对图像进行**有损压缩。**

使用 cmdline 的方式读取参数，参数格式为 {压缩指令([-compress] or [-read])} {待压缩文件名 or 待读取文件名}

对所提供的 lena.tiff 文件，程序运行结果如下：

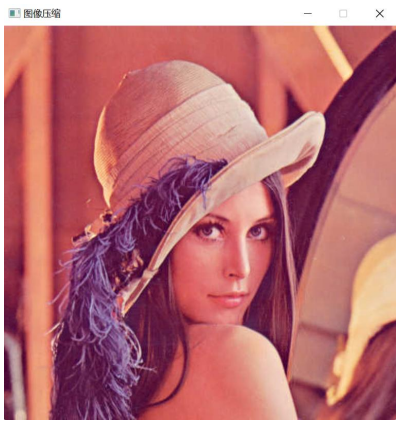
（1）指令为 -compress

```
C:\Users\DTTTTTTT\Desktop\图像压缩\x64\Debug>图像压缩.exe -compress lena.tiff
程序运行时间：1.038s
```

lena.jpg	2022/5/6 17:18	JPG 文件	31 KB
lena.tiff	2020/5/22 22:09	TIFF 图像	769 KB

（2）指令为 -read

```
C:\Users\DTTTTTTT\Desktop\图像压缩\x64\Debug>图像压缩.exe -read lena.jpg
show lena, press enter to continue...
Press enter to continue...
```



2. 设计思路:

总的来说，包括一下步骤：读取原图像并将图像分割成 8*8 的小块 -> 将 RGB 转为 YCbCr -> DCT（离散余弦变换） -> 数据量化 -> ZigZag 处理 ->

Huffman 编码 -> 写入文件。每个步骤的具体思路与程序设计如下：

（1）读取原图像(lena.tiff)，得到 8*8 小块的 YCbCr 值。

此步骤在函数【Compress】中实现。

第一步，用给定的 picReader 读取原图像（lena.tiff）的信息。

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

第二步，将图像分成 8*8 的小块，将 RGB 转换成 YCbCr，并将所有 8*8 小块的 YCbCr 值按照从上到下、从左到右的顺序存入 struct block(见下) 类型的**链表**中。

```
struct block {
    double Y[8][8], Cb[8][8], Cr[8][8];
    struct block* next;
}*head = NULL, * tail = NULL;
```

```
head = new(nothrow) block;
if (head == NULL) {
    cout << "内存申请失败" << endl;
    exit(0);
}
head->next = NULL;
tail = head;

for (int i = 0;i < x;i += 8)
    for (int j = 0;j < y;j += 8) {
        block* p = new(nothrow) block;
        if (p == NULL) {
            cout << "内存申请失败" << endl;
            exit(0);
        }
        tail->next = p, tail = p;

        for (int ki = i;ki < i + 8;ki++)
            for (int kj = j;kj < j + 8;kj++) {
                int cur = (ki * y + kj) * 4;
                p->Y[ki - i][kj - j] = 0.29871 * data[cur] + 0.58661 * data[cur + 1] + 0.11448 * data[cur + 2] - 128;
                p->Cb[ki - i][kj - j] = -0.16874 * data[cur] - 0.33126 * data[cur + 1] + 0.50000 * data[cur + 2];
                p->Cr[ki - i][kj - j] = 0.50000 * data[cur] - 0.41869 * data[cur + 1] - 0.08131 * data[cur + 2];
            }
        p->next = NULL;
    }

delete[] data;
data = nullptr;
```

（2）打印文件头

此步骤在函数【Get_ans(int im_height, int im_width)】中实现。

按照 JPEG 文件交换格式写就行，注意耐心细致就好，此处不再赘述。

（3）DCT（离散余弦变换）（注：从此步真正开始压缩图像信息!

此步骤在函数【DCT(double mt[8][8])】中实现。

离散余弦变换是图像压缩非常核心的一步，它能将矩阵中的数据集中到 DC 值（左上角第一个数据），以便于对 DC 值的部分进行压缩。

DCT 公式：

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

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

实现代码：

```
double alpha(int u) {
    if (u == 0)return 1.000 / sqrt(8);
    return 0.500;
}

void DCT(double mt[8][8]) {
    double f[8][8];
    for (int u = 0;u < 8;u++)
        for (int v = 0;v < 8;v++) {
            f[u][v] = 0;//init!
            for (int x = 0;x < 8;x++)
                for (int y = 0;y < 8;y++) {
                    f[u][v] += mt[x][y] * cos(1.0 * (2 * x + 1) / 16.0 * u * Pi) * cos(1.0 * (2 * y + 1) / 16.0 * v * Pi);
                }
            f[u][v] *= alpha(u) * alpha(v);
        }
    for (int i = 0;i < 8;i++) {
        for (int j = 0;j < 8;j++) {
            mt[i][j] = f[i][j];
        }
    }
}
```

（4）数据量化

此步骤在函数【quantify(int* tmp)】中实现。

在离散余弦变换后，虽然大部分 AC 值趋向于 0（与 DC 值相比），但仍然是繁琐的浮点数，为了进一步精简数据，需要进行数据量化。简单来说，就是用原矩阵的值除以某个合适系数再取整，使得大部分的 AC 值变为 0，使数据变得简洁。这一步也是压缩中**有损**的部分。

量化公式：

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

（其中，G 是原始数据，Q 是量化系数）

所使用的量化系数表：

```
const int Q[2][8][8] = { //量化表 Q[0]是L的 Q[1]是C的
  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,

  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,
```

实现代码：

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

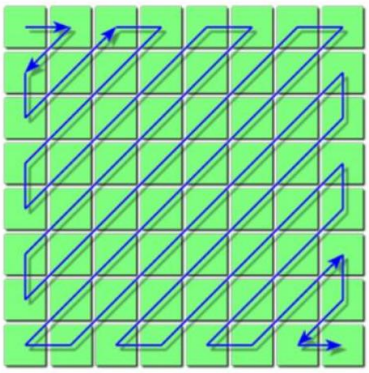
（注意到 fl 为 0 时表示对 Y 的数据量化处理，fl 为 1 时表示处理 Cb 或 Cr 的数据）

（5）ZigZag 处理（Z 字抖动）

此步骤在函数【zigzag(int* tmp)】中实现。

注意到进行数据量化之后，非零数据集中在矩阵的左上角，此时可以通过 ZigZag 处理将非零数据集中到矩阵的一维数组的前部，或者说将零集中到一维数组后部，便于后续压缩。

ZigZag 含义：将二维数组按照下图的顺序放入一维数组中。



实现代码：

```
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
};
```

```
void zigzag(int* tmp) {
    int ret[64];
    for (int i = 0;i < 64;i++)
        ret[ZigZag[i]] = tmp[i];
    for (int i = 0;i < 64;i++)
        tmp[i] = ret[i];
}
```

（6）编码：Rle + Bit + Huffman

此步骤在函数【Code(int *tmp, int type, int num)】中实现。

编码过程大致如下图：

<table>
<tr><td>①原始数据</td><td colspan="15">35,7,0,0,0,-6,-2,0,0,-9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,0,0,0,...,0</td></tr>
<tr><td rowspan="3">②RLE编码</td><td colspan="2">35</td><td colspan="2">7</td><td colspan="3">0,0,0,-6</td><td colspan="2">-2</td><td colspan="3">0,0,-9</td><td colspan="3">0,0,...,0,8</td><td>0,0,
...,0</td></tr>
<tr><td colspan="2">35</td><td colspan="2">7</td><td colspan="3">0,0,0,-6</td><td colspan="2">-2</td><td colspan="3">0,0,-9</td><td colspan="2">0,0,...,0</td><td colspan="2">0,0,8</td><td>0,0,
...,0</td></tr>
<tr><td>(0,35)</td><td>(0,7)</td><td colspan="3">(3,-6)</td><td colspan="2">(0,-2)</td><td colspan="3">(2,-9)</td><td colspan="2">(15,0)</td><td colspan="2">(2,8)</td><td>EOB</td></tr>
<tr><td>③BIT编码</td><td>(0,6, 100011)</td><td>(0,3, 111)</td><td colspan="3">(3,3, 001)</td><td colspan="2">(0,2, 01)</td><td colspan="3">(2,4, 0110)</td><td colspan="2">(15,-)</td><td colspan="2">(2,4, 1000)</td><td>EOB</td></tr>
<tr><td>④哈弗曼编码</td><td>(0x6,100011)</td><td>(0x3,111)</td><td colspan="3">(0x33,001)</td><td colspan="2">(0x2,01)</td><td colspan="3">(0x24,0110)</td><td colspan="2">0xF0</td><td colspan="2">(0x24,1000)</td><td>EOB</td></tr>
<tr><td>④哈弗曼编码</td><td>100</td><td>100011</td><td>100</td><td>111</td><td colspan="2">1111 1111 0101</td><td>001</td><td>01</td><td>01</td><td colspan="2">1111 1111 0100</td><td>0110</td><td colspan="2">1111 1111 001</td><td colspan="2">1111 1111 0100</td><td>1000</td><td>1010</td></tr>
<tr><td rowspan="2">⑤序列化</td><td colspan="16">10010001110011111111111101010010101111111110100011011111111001111111110100100010 10</td></tr>
<tr><td colspan="16">91 CF FE A5 7F D1 BF CF FA 45</td></tr>
</table>

就具体的代码实现来说：

第一步 对矩阵的第一个值（DC 值）进行特殊处理：减去前一个小块对应 DC 值的原值，变为偏移值。

第二步 统计尾部的 0，找到从后往前数第一个非 0 值的位置 pos。（注意此处统计仅限于数组的第 2~64 个值，不能包括第一个值（DC 偏移值））

第三步 对于第 1 个到第 pos 个数据，先进行 RLE 编码，找到每一段的前置 0 数量（zero_num）与末尾值（tmp[i]），将其 BIT 编码的前半部分对应的哈夫曼编码和后半部分（BIT 编码）加入字符串 str。

第四步 若有尾部 0，在 str 后加上 eof 对应的编码。

第五步 通过前面四个步骤可以得到一个 01 串，现在需要将 01 串分成 8 个一组，转成相应的 unsigned char 类型存入答案字符串 ans 中。

```
void Code(int* tmp, int type, int num) {

    int tmp0 = tmp[0]; //对第一个值（DC）进行特殊处理：减去前一个小块该值的原值
    tmp[0] -= lastone[num];
    lastone[num] = tmp0;

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

    //开始编码
    int zero_num = 0;
    bool is_AC = 0;
    string str = leftstr;
    for (int i = 0;i < 64 - tail_zero;i++) {
        if (tmp[i] != 0 || zero_num == 15 || i == 0) { //三种情况缺一不可
            int flag = 0;
            if (tmp[i] < 0) flag = 1, tmp[i] *= -1;
            int bin_len = GetBinLen(tmp[i]);
            bitset<40>bit(tmp[i]);
            str = str + Huffcode[type + is_AC][(zero_num << 4) | bin_len];
            for (int i = bin_len - 1;i >= 0;i--)
                if (bit[i] ^ flag) { str = str + '1'; }
                else { str = str + '0'; }
            zero_num = 0;
        }
        else
            zero_num++;
        is_AC = 1;
    }

    if (tail_zero) str = str + Huffcode[type + 1][0];//eob！并不都是1010
```

注意到在编码前，应该根据给定的哈夫曼参数表预处理出所有 value 对应的哈夫曼编码：

```
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 Get_Huffman_Code(const char* Numbers, const unsigned char* Values, int ind) {
    const unsigned char* values = Values;
    int code = 0;
    for (int i = 0;i < 16;i++) {
        char num = Numbers[i];
        int n_bits = i + 1;
        for (int j = 0;j < num;j++) {
            unsigned char value = *values;
            string str;
            bitset<50>bit(code);
            for (int k = i;k >= 0;k--) {
                if (bit[k] == 0) str = str + '0';
                else str = str + '1';
            }
            Huffcode[ind][value] = str;
            code++;
            values++;
        }
        code <<= 1;
    }
}
```

（7）写文件

此步骤在函数【Compress(const char *infile)】中实现。

由于在写文件头与压缩图像信息时以及将信息都写入答案字符串 ans 中，所以直接向目标文件（lena.jpg）中输出答案字符串 ans 就行。

二．在实验过程中遇到的困难及解决方法

(ps: 感觉 1.和 3.不完全算是困难，主要是关于如何巧妙简化代码的设计吧(emm 就是个人比较沉迷于这种小设计))

1．**困难**： 预处理出 value 对应的哈夫曼码如何存储能方便调用？

解决方法：显而易见的一点是将哈夫曼码存入<unsigned char, string>类型的

unordered_map 中。那么如何存来方便调用呢?

```
unordered_map<unsigned char, string>Huffcode[4];
```

```
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);
}
```

将 L 的 DC 与 AC 相应的哈夫曼码分别存入 Huffman[0]与 Huffman[1]，将 C 的 DC 与 AC 相应的哈夫曼码分别存入 Huffman[2]与 Huffman[3]。**规律**：在 L 在前 C 在后的前提下，DC 在前，AC 在后。另外，在函数【Code】中增加参数 type 与 bool 类型的 is_AC 变量。

为什么这样就方便调用了呢?

具体来说, 当处理L数据时, type=0, 初始化is_AC=0; 处理C数据时, type=1, 同样初始化 is_AC=0。这样一来，**type + is_DC 的值就代表了要用的 Huffman 表是哪个，非常简洁而无需繁琐的分类讨论。**

```
str = str + Huffcode[type + is_AC][(zero_num << 4) | bin_len];
```

（相似的用一个简洁的式子来代替分类讨论的方法也用于数据量化选择量化表的部分与[困难 2]）

2. **困难**：如何方便地得到一个数的二进制?

解决方法：**bitset!** 能够方便地实现数与其二进制的双向转换，不过需要注意到 bitset 中存入表示二进制的 string 时，低位在前高位在后。

3. **困难**：在从 RLE 到 BIT 的过程中，如何统一 RLE 的一段的尾数为正与负的情况?

解决方法：首先需要注意到负数的 BIT 编码时其绝对值的 BIT 编码取反，并非负数的二进制表示。此处用一个 flag 来标记是否为负数（正数为 0，负数为 1），并且将该数取绝对值，用 bitset 得到该绝对值对应的二进制表示之后，在逐位将

二进制表示复制到序列字符串 str 的过程中，将单纯的若 bitset 该位的值为 1 则 str=str+'1'的判定 改为 若(bit[i] ^ flag)为 1 则 str=str+'1'。如此便可简洁实现统一而无需分类讨论。

4. **困难**：发现程序运行地非常慢，需要 10s 的时间来压缩 lena，而认识的同学只花了 2s。

解决思路：考虑到算法流程都是相同的，想到是输出的问题。最开始输出的方法是逐个输出每个 unsigned char，为了简洁使用，将输出写成一个单独的函数。这不仅意味着每个字符都是单独输出的，也意味着文件再不断地打开关闭，因为每次输出一个字符，都要开关一次文件，耗费了大量时间。

解决思路：将所有的 unsigned char 连成 string，然后一次性输出，只打开和关闭文件一次。

关于问题的一个小思考：到底是开关文件耗费了大量时间还是逐个输出字符耗费了大量时间呢？做控制变量的实验就可以得到这个问题的答案。发现如果对于每个字符都打开关闭一次文件（不做写操作），程序运行需要 4s，也就是说开关文件约耗费 3s，逐个输出字符约耗费 6s。

```
C:\Users\DTTTTTTT\Desktop\图像压缩\x64\Debug>图像压缩.exe -compress lena.tiff
程序运行时间：10.034s
```

```
C:\Users\DTTTTTTT\Desktop\图像压缩\x64\Debug>图像压缩.exe -compress lena.tiff
程序运行时间：4.241s
```

5. **困难**：对着程序左看右看也 de 不出 bug 怎么办？

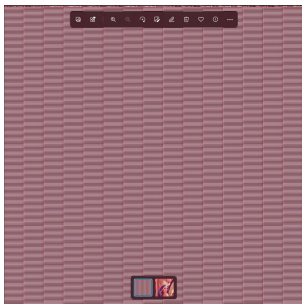
解决方法：

（1）如果当前图片和 lena 没点关系

找大佬发一张 ta 以及压缩好的 Lena，用 notepad++的 HEX Editor 打开 ta 的 lena 和我的乱码图片，开始**肉眼核对与地毯式查错**。如果除开文件头，图像信息的内容只需要对前几行就可以发现大部分的 bug（包括手误写错的地方和对算法细节理解有误的地方），然后得到一张至少和 lena 有那么一点沾边的图片。如果还不对，就去高程群里问问 :) 亲测有效。

（2）如果当前图片和 lena 至少有那么些关系了

可以静下来尝试自己推理下可能出错的地方。比如只显示处理 lena 的帽子，帽子以下的图片没有显示，可以想到可能提前终止了读取图片信息，也就是说可能在图片信息中出现了 EOI 标识符 0xFFD9，意识到问题所在了就解决了问题的一大半了。比如只有顶上几行和 lena 有关系，顺序很乱，但隐约能看到一些形状（如下图），可以想到可能是图片的长和宽（分辨率）输错了。



放大后：

~~三.一踩过的坑 一(正片开始)~~

1. 文件头

(1) 量化表也需要 zigzag 输出!

(2) 颜色分量的 ID 只能是 01 02 03 而不能是 00 01 02!

2. 离散余弦处理和量化、zigzag 处理完毕后，需要对序列的第一个值（DC 值）进行**偏移**，即减去前一个小块对应序列的 DC **原值**。

3. RLE 过程中：**DC 值一定要单独显示出来。**换句话说，RLE 的第一个小段一定是（0,x）。若 DC 值为 0，它既不能归为前缀的 0，也不能计入尾部的 0，具体来说如以下两个例子：

(1) 若序列全为 0 ：其 RLE 并非 EOF，而是(0,0)、EOF

(2) 若序列为 0 0 0 1 0 0 0 0 0 0：其 RLE 并非(3,1)、EOF，而是(0,0)、(2,1)、EOF

4. **EOB 对应的哈夫曼码**并不都是 1010，而应该是 0 对应的相应哈夫曼表的编码。对 L 来说是 1010，但对 C 来说是 00。

5. **0 对应的 BIT 码**不是 0 而是空！注意观察 BIT 对应表的细节

6. 在**预处理哈夫曼码**时，不能将其存入 unordered_map<unsigned char, int> 中，因为**哈夫曼码中的前置 0 是不可省略**的，若将其转化成整数存入 int 中会丢失前置 0 的信息。

7. 还有一些 把改写 Cr Cb 的地方都写成 Y（复制粘贴没改全）、计算二进制长度的函数写错等等等等智障错误 qaq。真・活的 bug 制造机。

四．心得体会

1. 还是注意细节（每次都要强调的!

（1）打开文件后，检查文件是否成功打开

（2）打开的文件都要及时关闭

（3）申请的动态内存都要及时处理：

结束操作后，需要 delete 掉所有的链表结点和读取图像用的 BYTE* data。

2. 一些感受（碎碎念

这次作业总共花了七天时间，其中比较核心的部分写了五天，后两天是悠闲

地做一些收尾工作与写作业报告。

最开始听学长的分享课，说实话只听懂了分割图片，后面都是一脸懵（感觉真的需要一些预习）。因为 yfjj 提醒过这个作业难度很大，所以很早就开始了。最初对着 PPT 和 blog 自学，感觉看了很久还是不怎么理解，陷入了一些绝望和焦虑的情绪。半天之后还是意识到要冷静，于是又耐心仔细地学 PPT 和 blog 的内容，反复地看，每一次看都能多懂一点。以及，把抽象地焦虑变成提出具体的问题并通过各种方式得到答案。最终对整个过程有了相对清晰的认识，对程序的流程也有了总体的一些规划才终于开始 code。

我感觉写代码对我来说快乐的部分主要在于两点：一是学习具体的知识；二是设计程序中的逻辑：按照怎样的形式来实现算法？有没有什么办法来简化代码（如[二]中的困难 1，3）？等等。最痛~~并快乐着~~的部分当然是 debug（痛苦面具）。这次作业自行设计的部分并不太多（与上次的文件压缩相比）而坑却有点多，所以就显得这次作业格外痛苦了。

这次作业实在能称得上一波三折和命途多舛，写作业的心情也是落起落落落落落落落起落落落落落落落落落落落落落落落落落落落落落落落落落起。但是真的是非常珍贵的经历呐。

3. 小小的建议

也许能对每次大作业开个共享文档，大家集中写下自己踩到的（非个性化错误）坑，这样就不会每个人都在每个坑里摸爬滚打了叭 :)

也许能在大作业分享课之前先发下 PPT，来不及的话发点要讲的算法相关的参考资料，或者只是预告一下要讲啥算法，这样能提前预习一下 qaq

五. 源代码

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

```
#include<time.h>

using namespace std;

//一些需要用到的常量----------------------------------------------------------------
const double Pi = 3.14159;
const int Q[2][8][8] = { //量化表 Q[0]是 L 的 Q[1]是 C 的
    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,

    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 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
};
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 };
const char Standard_DC_Chrominance_NRCodes[] = { 0, 3, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0 };
const unsigned char Standard_DC_Chrominance_Values[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
const char Standard_AC_Luminance_NRCodes[] = { 0, 2, 1, 3, 3, 2, 4, 3, 5, 5, 4, 4, 0, 0, 1, 0x7d };
const unsigned char 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
};
const char Standard_AC_Chrominance_NRCodes[] = { 0, 2, 1, 2, 4, 4, 3, 4, 7, 5, 4, 4, 0, 1, 2, 0x77 };
const unsigned char 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
};
```

```
96.  struct block {
97.      double Y[8][8], Cb[8][8], Cr[8][8];
98.      struct block* next;
99.  }*head = NULL, * tail = NULL;
100.
101. unordered_map<unsigned char, string>Huffcode[4];
102. string ans; //答案字符串
103.
104. double alpha(int u) {
105.     if (u == 0)return 1.000 / sqrt(8);
106.     return 0.500;
107. }
108.
109. void DCT(double mt[8][8]) {
110.     double f[8][8];
111.     for (int u = 0;u < 8;u++)
112.         for (int v = 0;v < 8;v++) {
113.             f[u][v] = 0;//init!
114.             for (int x = 0;x < 8;x++)
115.                 for (int y = 0;y < 8;y++) {
116.                     f[u][v] += mt[x][y] * cos(1.0 * (2 * x + 1) / 16.0 * u * Pi) * cos(1.0 * (2 * y + 1) / 16.0 * v * Pi);
117.                 }
118.             f[u][v] *= alpha(u) * alpha(v);
119.         }
120.     for (int i = 0;i < 8;i++) {
121.         for (int j = 0;j < 8;j++) {
122.             mt[i][j] = f[i][j];
123.         }
124.     }
125. }
126.
127. void quantify(double mt[8][8], bool fl, int* tmp) {
128.     for (int i = 0;i < 8;i++)
129.         for (int j = 0;j < 8;j++) {
130.             tmp[i * 8 + j] = round(mt[i][j] / Q[fl][i][j]);
131.         }
132. }
133.
134. void zigzag(int* tmp) {
135.     int ret[64];
136.     for (int i = 0;i < 64;i++)
137.         ret[ZigZag[i]] = tmp[i];
138.     for (int i = 0;i < 64;i++)
139.         tmp[i] = ret[i];
```

```
}

int GetBinLen(int x) {
    if (x == 0) return 0;
    if (x < 0) x = -x;
    for (int i = 31;i >= 0;i--) {
        if (x & (1 << i)) return i + 1;
    }
}

string leftstr;
int lastone[3];

void Code(int* tmp, int type, int num) {

    int tmp0 = tmp[0]; //对第一个值（DC）进行特殊处理：减去前一个小块该值的原值
    tmp[0] -= lastone[num];
    lastone[num] = tmp0;

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

    //开始编码
    int zero_num = 0;
    bool is_AC = 0;
    string str = leftstr;
    for (int i = 0;i < 64 - tail_zero;i++) {
        if (tmp[i] != 0 || zero_num == 15 || i == 0) { //三种情况缺一不可
            int flag = 0;
            if (tmp[i] < 0) flag = 1, tmp[i] *= -1;
            int bin_len = GetBinLen(tmp[i]);
            bitset<40>bit(tmp[i]);
            str = str + Huffcode[type + is_AC][(zero_num << 4) | bin_len];
            for (int i = bin_len - 1;i >= 0;i--)
                if (bit[i] ^ flag) { str = str + '1'; }
                else { str = str + '0'; }
            zero_num = 0;
        }
        else
            zero_num++;
        is_AC = 1;
```

```
184.     }
185.
186.     if (tail_zero) str = str + Huffcode[type + 1][0];//eob！并不都是 1010
187.
188.     int cur = 0;
189.     while (str.length() > cur + 7) {
190.         bitset<8>bit(str.substr(cur, 8));
191.         cur += 8;
192.         unsigned char c = bit.to_ullong();
193.         ans.push_back(c);
194.         if (c == 0xff)  ans.push_back(0x00);
195.     }
196.     leftstr = str.substr(cur, str.length() - cur);
197. }
198.
199. void Get_Huffman_Code(const char* Numbers, const unsigned char* Values, int ind) {
200.     const unsigned char* values = Values;
201.     int code = 0;
202.     for (int i = 0;i < 16;i++) {
203.         char num = Numbers[i];
204.         int n_bits = i + 1;
205.         for (int j = 0;j < num;j++) {
206.             unsigned char value = *values;
207.             string str;
208.             bitset<50>bit(code);
209.             for (int k = i;k >= 0;k--) {
210.                 if (bit[k] == 0) str = str + '0';
211.                 else str = str + '1';
212.             }
213.             Huffcode[ind][value] = str;
214.             code++;
215.             values++;
216.         }
217.         code <<= 1;
218.     }
219. }
220.
221. void Init_Huffman_Code() {
222.     Get_Huffman_Code(Standard_DC_Luminance_NRCodes, Standard_DC_Luminance_Values, 0);
223.     Get_Huffman_Code(Standard_AC_Luminance_NRCodes, Standard_AC_Luminance_Values, 1);
224.     Get_Huffman_Code(Standard_DC_Chrominance_NRCodes, Standard_DC_Chrominance_Values, 2);
225.     Get_Huffman_Code(Standard_AC_Chrominance_NRCodes, Standard_AC_Chrominance_Values, 3);
226. }
227.
```

```
228.  void Get_ans(int im_height, int im_width) {
229.
230.      //SOI Start of Image 图像开始------------------------------------------------------------
231.      ans.push_back(0xff);ans.push_back(0xd8);  //标记代码 0xffd8
232.
233.      //APP0 Application-----------------------------------------------------------------------
234.      ans.push_back(0xff);ans.push_back(0xe0); //标记代码 0xffe0
235.      ans.push_back(0x00);ans.push_back(0x10); //16！数据长度
236.                  ans.push_back(0x4a);ans.push_back(0x46);ans.push_back(0x49);ans.push_back(0x46);ans.push_back(0x00);//  标  识
    符 0x4A46494600
237.      ans.push_back(0x01);ans.push_back(0x02);//版本号 0x0102
238.      ans.push_back(0x00);//X 和 Y 的密度单位
239.      ans.push_back(0x00);ans.push_back(0x01);//X 方向像素密度
240.      ans.push_back(0x00);ans.push_back(0x01); //Y 方向像素密度
241.      ans.push_back(0x00); //缩略图水平像素数目
242.      ans.push_back(0x00);//缩略图垂直像素数目
243.
244.      //DQT Define Quantization Table 定义量化表-----------------------------------------------
245.      ans.push_back(0xff);ans.push_back(0xdb); //标记代码 0xffdb
246.      ans.push_back(0x00);ans.push_back(0x84); //132！数据长度
247.      ans.push_back(0x00); //第 0 个量化表 是 L 的量化表！注意量化表的输出也要 zigzag
248.      int tmp[64], tmp1[64];
249.      for (int i = 0;i < 8;i++)
250.          for (int j = 0;j < 8;j++)
251.              tmp[i * 8 + j] = Q[0][i][j];
252.      for (int i = 0;i < 64;i++)tmp1[ZigZag[i]] = tmp[i];
253.      for (int i = 0;i < 64;i++) ans.push_back(tmp1[i]);
254.      ans.push_back(0x01); //第 1 个量化表 是 C 的量化表！
255.      for (int i = 0;i < 8;i++)
256.          for (int j = 0;j < 8;j++)
257.              tmp[i * 8 + j] = Q[1][i][j];
258.      for (int i = 0;i < 64;i++)tmp1[ZigZag[i]] = tmp[i];
259.      for (int i = 0;i < 64;i++) ans.push_back(tmp1[i]);
260.
261.      //SOF0 Start of Frame 帧图像开始---------------------------------------------------------
262.      ans.push_back(0xff);ans.push_back(0xc0); //标记代码 0xffc0
263.      ans.push_back(0x00);ans.push_back(0x11); //17！数据长度
264.      ans.push_back(0x08); //8 位 精度
265.      ans.push_back(im_height >> 8);ans.push_back(im_height & 15); //图像高度
266.      ans.push_back(im_width >> 8);ans.push_back(im_width & 15);// 图像宽度
267.      ans.push_back(0x03); // 颜色分量数
268.      ans.push_back(0x01);ans.push_back(0x11);ans.push_back(0x00); //颜色分量 ID (0 Y 通道) 水平垂直采样因子 量化表 ID
269.      ans.push_back(0x02);ans.push_back(0x11);ans.push_back(0x01); //颜色分量 ID(1 Cr 通道) 水平垂直采样因子 量化表 ID
270.      ans.push_back(0x03);ans.push_back(0x11);ans.push_back(0x01); //颜色分量 ID(1 Cb 通道) 水平垂直采样因子 量化表 ID
```

```
271.
272.    //DHT Define Huffman Table 定义哈夫曼表----------------------------------------------------------------
273.    ans.push_back(0xff);ans.push_back(0xc4); //标记代码 0xffc4
274.    ans.push_back(0x01);ans.push_back(0xa2);//418！数据长度
275.    ans.push_back(0x00); //DC 第 0 个哈夫曼表（L
276.    for (int i = 0;i < 16;i++)
277.        ans.push_back(Standard_DC_Luminance_NRCodes[i]);
278.    for (int i = 0;i < 12;i++)
279.        ans.push_back(Standard_DC_Luminance_Values[i]);
280.    ans.push_back(0x10); //AC 第 0 个哈夫曼表（L
281.    for (int i = 0;i < 16;i++)
282.        ans.push_back(Standard_AC_Luminance_NRCodes[i]);
283.    for (int i = 0;i < 162;i++)
284.        ans.push_back(Standard_AC_Luminance_Values[i]);
285.    ans.push_back(0x01); //DC 第 1 个哈夫曼表(C
286.    for (int i = 0;i < 16;i++)
287.        ans.push_back(Standard_DC_Chrominance_NRCodes[i]);
288.    for (int i = 0;i < 12;i++)
289.        ans.push_back(Standard_DC_Chrominance_Values[i]);
290.    ans.push_back(0x11); //AC 第 1 个哈夫曼表 (C
291.    for (int i = 0;i < 16;i++)
292.        ans.push_back(Standard_AC_Chrominance_NRCodes[i]);
293.    for (int i = 0;i < 162;i++)
294.        ans.push_back(Standard_AC_Chrominance_Values[i]);
295.
296.    //SOS Start of Scan 扫描开始----------------------------------------------------------------------
297.    ans.push_back(0xff);ans.push_back(0xda); //标记代码 0xffda
298.    ans.push_back(0x00);ans.push_back(0x0c);//12! 数据长度
299.    ans.push_back(0x03); //颜色分量数
300.    ans.push_back(0x01);ans.push_back(0x00); //Y 分量 直流 0 交流 0
301.    ans.push_back(0x02);ans.push_back(0x11); //Cr 分量 直流 1 交流 1
302.    ans.push_back(0x03);ans.push_back(0x11); //Cb 分量 直流 1 交流 1
303.    ans.push_back(0x00);ans.push_back(0x3f);ans.push_back(0x00);
304.
305.    //图像信息----------------------------------------------------------------------------------------
306.    block* p = head;
307.    while (p->next != NULL) {
308.        p = p->next;
309.        DCT(p->Y);
310.        int tmp2[64];
311.        quantify(p->Y, 0, tmp2);
312.        zigzag(tmp2);
313.        Code(tmp2, 0, 0);
314.
```

```
            DCT(p->Cb);
            quantify(p->Cb, 1, tmp2);
            zigzag(tmp2);
            Code(tmp2, 2, 1);

            DCT(p->Cr);
            quantify(p->Cr, 1, tmp2);
            zigzag(tmp2);
            Code(tmp2, 2, 2);
        }

        if (leftstr.length()) {
            int addzero = (8 - leftstr.length() % 8) % 8;
            for (int i = 0;i < addzero;i++)leftstr = leftstr + '0';
            bitset<8>bit(leftstr);
            unsigned char c = bit.to_ulong();
            ans.push_back(c);
            if (c == 0xff)  ans.push_back(0x00);
        }

        //EOI End of Image 扫描结束----------------------------------------------------------------------------
        ans.push_back(0xff);ans.push_back(0xd9);
    }

    void Delete() {
        block* p = head;
        while (p != NULL) {
            block* q = p->next;
            delete p;
            p = q;
        }
    }

    void Compress(const char* infile) {
        //预处理出所有的哈夫曼编码并存入 unordered_map<unsigned char,string> Huffcode[4];中
        Init_Huffman_Code();

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

        head = new(nothrow) block;
```

```
359.     if (head == NULL) {
360.         cout << "内存申请失败" << endl;
361.         exit(0);
362.     }
363.     head->next = NULL;
364.     tail = head;
365.
366.     for (int i = 0;i < x;i += 8)
367.         for (int j = 0;j < y;j += 8) {
368.             block* p = new(nothrow) block;
369.             if (p == NULL) {
370.                 cout << "内存申请失败" << endl;
371.                 exit(0);
372.             }
373.             tail->next = p, tail = p;
374.
375.             for (int ki = i;ki < i + 8;ki++)
376.                 for (int kj = j;kj < j + 8;kj++) {
377.                     int cur = (ki * y + kj) * 4;
378.                     p->Y[ki - i][kj - j] = 0.29871 * data[cur] + 0.58661 * data[cur + 1] + 0.11448 * data[cur + 2] - 128;
379.                     p->Cb[ki - i][kj - j] = -0.16874 * data[cur] - 0.33126 * data[cur + 1] + 0.50000 * data[cur + 2];
380.                     p->Cr[ki - i][kj - j] = 0.50000 * data[cur] - 0.41869 * data[cur + 1] - 0.08131 * data[cur + 2];
381.                 }
382.             p->next = NULL;
383.         }
384.
385.     delete[] data;
386.     data = nullptr;
387.
388.     //打印文件头，压缩图像并输出至 lena.jpg
389.     Get_ans(x, y);
390.
391.     //输出
392.     const char* outfile = "lena.jpg";
393.     ofstream fl(outfile, ios::binary);
394.     if (!fl) {
395.         cout << "文件打开失败" << endl;
396.         exit(0);
397.     }
398.     fl << ans;
399.     fl.close();
400.
401.     //释放动态申请的空间（链表）
402.     Delete();
```

```
}
void Show(const char* readfile) {
    printf("show lena, press enter to continue...");
    (void)getchar();

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

    delete[] data;
    data = nullptr;
    printf("Press enter to continue...");
    (void)getchar();
}
int main(int argc, char** argv) {
    clock_t starttime, endtime;
    starttime = clock();

    if (argc != 3) {
        cerr << "Please make sure the number of parameters is correct." << endl;
        return -1;
    }

    if (strcmp(argv[1], "-compress") == 0) {
        Compress(argv[2]);
    }
    else if (strcmp(argv[1], "-read") == 0) {
        Show(argv[2]);
    }
    else {
        cerr << "Unknown parameter!\nCommand list:\nzip/unzip" << endl;
        return -1;
    }

    endtime = clock();
    cerr << "程序运行时间：" << 1.00 * (endtime - starttime) / CLOCKS_PER_SEC << "s" << endl;

    return 0;
}
```

完结撒花！