实 验 报 告

实验题目： 编写基于Huffman算法的无损压缩程序和解压程序

姓名：黄玟瑜

学号：19335074

日期：2020年10月24日

院（系）：数据科学与计算机学院

专业（班级）： 计算机科学与技术（超算）

1. **实验目的**

编写基于Huffman算法的压缩和解压缩程序。你的程序应该能够压缩任意文件，并能无损解压。

**二. 实验环境**

本实验可基于Visual Studio Code等平台开发，参考主流的编码规范，如Google C++Style Guide（中文版）

**2.1 编程语言和开发工具**

编程语言： ANSI C/C++

开发工具： Visual Studio Code、编译器G++

**2.2 编码规范**

遵循良好的程序设计风格来设计和编写程序。基本编码规范：

1. 标识符的命名要到达顾名思义的程度；

2. 关键代码提供清晰、准确的注释；

3. 程序版面要求：

a) 不同功能块用空行分隔；

b) 一般一个语句一行；

c) 语句缩进整齐、层次分明。

**三、实验内容**

压缩文件：根据ASCII码文件中各ASCII字符出现的频率情况创建Huffman树，再将各字符对应的哈夫曼编码写入压缩文件中，实现文件压缩。

解压文件：根据压缩文件时 创建的Huffman树构建解码词典，解码词典从压缩文件中读取Huffman编码并翻译成对应字符写入解压文件中，实现文件解压。

1. **分析与设计**

**实验原理：**

Huffman编码的目的是，如何用更短的bit来编码数据。

通过变长编码压缩编码长度。我们知道普通的编码都是定长的，比如常用的ASCII编码,每个字符都是8个bit。但在很多情况下，数据文件中的字符出现的概率是不均匀的，比如在一篇英语文章中，字母“E”出现的频率最高，“Z”最低，这时我们可以使用不定长的bit编码，频率高的字母用比较短的编码表示，频率低的字母用长的编码表示。

但这就要求编码要符合“前缀编码”的要求，即较短的编码不能是任何较长的编码的前缀，这样解析的时候才不会混淆。要生成这种编码，最方便的就是用二叉树，把要编码的字符放在二叉树的叶子上，所有的左节点是0，右节点是1，从根浏览到叶子上，因为字符只能出现在树叶上，任何一个字符的路径都不会是另一字符路径的前缀路径，符合前缀原则编码就可以得到。

1.Huffman树和Huffman算法（核心思想）

定义：给定N个权值作为N个叶子结点，构造一棵二叉树，若该树的带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。

构建Huffman树：

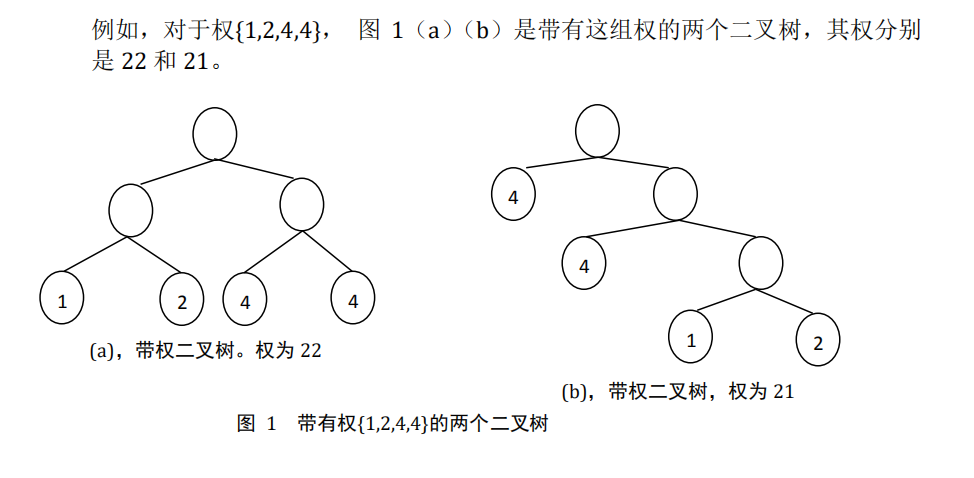
假设有n个权值，则构造出的哈夫曼树有n个叶子结点。 n个权值分别设为 w1、w2、…、wn，则哈夫曼树的构造规则为：

(1) 将w1、w2、…，wn看成是有n 棵树的森林(每棵树仅有一个结点)；

(2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

(3)从森林中删除选取的两棵树，并将新树加入森林；

(4)重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。



2. 前缀码

定义：任意一个字符的编码都不是另一个字符的编码的前缀,这种编码叫做前缀编码。

由于Huffman编码的这种特性，在解压缩时不会混淆。

3. 文件和文件流

文件：内存中存放的数据在计算机关机后就会消失。要长久保存数据，就要使用硬盘、光盘、U 盘等设备。为了便于数据的管理和检索，引入了“文件”的概念。一般来说，文件可分为文本文件、视频文件、音频文件、图像文件、可执行文件等多种类别，这是从文件的功能进行分类的。从数据存储的角度来说，所有的文件本质上都是一样的，都是由一个个字节组成的，归根到底都是 0、1 比特串。不同的文件呈现出不同的形态（有的是文本，有的是视频等等），这主要是文件的创建者和解释者（使用文件的软件）约定好了文件格式。

将文件中定长的字节编码成不定长的Huffman编码，使在文件在数据不丢失的情况下文件的长度变短，从而实验压缩文件的目的。

**设计程序：**

压缩文件

解压文件

函数设计：

Huffman.hpp

#include <iostream>

#include <map>

#include <bitset>

#include <string>

#include <cstring>

#include <queue>

#include <list>

#include <fstream>

#include <iomanip>

using **namespace** std;

**struct** tree\_node{

**unsigned** **char** name;

    tree\_node \*left, \*right;

**size\_t** weight;

    string code;

    tree\_node(**unsigned** **char** name\_, tree\_node **\***left\_, tree\_node **\***right\_,  **size\_t** weight\_, string code\_){

        name = name\_;*//字符*

        left = left\_;

        right = right\_;

        weight = weight\_;*//权重（字符出现频率）*

        code = code\_;*//字符对应的Huffman编码的string形式*

    }

};

*//统计文件长度，计算各字符出现的频率，返回文件长度，字符到频率的映射存储在char\_weight中*

**size\_t** char\_account(ifstream **&**input, map<**char**, **size\_t**> **&**char\_weight);

*//跟据字符到频率的映射构建半哈夫曼树，返回树的根节点,此时哈夫曼树未编码*

tree\_node**\*** built\_tree(map<**char**, **size\_t**> **&**char\_weight);

*//销毁树，懂得都懂*

**void** destroy\_tree(tree\_node **\***);

*//根据半哈夫曼树（节点未编码）得到完整的哈夫曼树 （在结点处得到字符到它对应的哈夫曼编码和映射*

**void** encoder(tree\_node **\***root, map<**char**, string> **&**dictionary);

*//压缩过程，压缩成功返回TRUE*

**bool** compress(**const** string **&**source\_filename, **const** string **&**encoded\_filename);

*//解压过程，压缩成功返回TRUE*

**bool** uncompress(**const** string **&**encoded\_filename, **const** string **&**source\_filename);

*//统计文件长度*

**size\_t** get\_filesize(**const** string **&**filename);

//修改文件拓展名

string change\_extension(**const** string**&** filename, **const** string**&** extension);

main.cpp

#include "Huffman.hpp"

#define std\_extension "hwy"*//自定义压缩文件的拓展名*

string change\_extension(**const** string**&** filename, **const** string**&** extension){

**size\_t** p = filename.find\_last\_of('.');

    string str = filename.substr(0, p) + "." + extension;

    return str;

}

**int** main(){

    string input\_filename, output\_filename, ex;

**int** select;

    do{

        cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

        cout << "                           1: Compress" << endl;

        cout << "                           2: Uncompress" << endl;

        cout << "                           0: Exit" << endl;

        cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

        cout << "   Your choice:" << endl;

        cin >> select;

        switch(select){

            case 1:

                cout << "Please enter the name of the file you want to compress:" << endl;

                cin >> input\_filename;

*//压缩文件名和原始文件名相同，拓展名为自定义的拓展名*

                output\_filename = change\_extension(input\_filename,std\_extension);

*//压缩过程*

*//汇报压缩信息*

                if(compress(input\_filename,output\_filename)){

                    cout << "Compress successfully!" << endl;

                    cout << "Size of input file: " << get\_filesize(input\_filename) << endl;

                    cout << "Size of output file: " << get\_filesize(output\_filename) << endl;

                    if(get\_filesize(input\_filename)>0)

                        cout << "Compress ratio: " << ((**double**)get\_filesize(output\_filename) / (**double**)get\_filesize(input\_filename)) \* 100 << "%" << endl;

                }

                else

                {

                    cout << "Fail to compress." << endl;

                }

                break;

            case 2:

                cout << "Please enter the name of the file you want to uncompress:" << endl;

                cin >> input\_filename;

*//拓展名和自定义拓展名不同的文件不能解压*

                if(input\_filename.substr(input\_filename.find\_last\_of('.')+1) != std\_extension){

                    cout << "This kind of file can not be uncompressed!" << endl;

                    break;

                }

*//输入要解压文件的格式*

                cout << "Please enter the extension of the output file:" << endl;

                cin >> ex;

                output\_filename = change\_extension(input\_filename,ex);

*//解压过程*

*//汇报解压信息*

                if(uncompress(input\_filename,output\_filename)){

                    cout << "Uncompress successfully!" << endl;

                    cout << "Size of input file: " << get\_filesize(input\_filename) << endl;

                    cout << "Size of output file: " << get\_filesize(output\_filename) << endl;

                }

                else

                {

                    cout << "Fail to uncompress." << endl;

                }

                break;

            case 0:

                break;

            default:

                cout << "Invalid choice! Please try again." << endl;

        }

        input\_filename.clear();

        output\_filename.clear();

        cout << endl;

    } while (select);

    cout << "Exit successfully!" << endl;

    system("pause");

    return 0;

}

1、用户可以选择压缩或解压缩文件，或退出系统

2、压缩文件的格式都统一为自定义的格式，系统默认无法识别，防止文件被错误打开

3、使用分界，使程序界面整洁美观

4、汇报压缩解压信息和压缩比率，使用户对压缩情况一目了然

Huffman.cpp

该文件包含各个函数的实现，全部贴出过于冗长，请自行查看代码，其中包含了注释。

下面压缩过程和解压过程

**bool** uncompress(**const** string **&**encoded\_filename, **const** string **&**source\_filename){

    ifstream in(encoded\_filename.c\_str(), ios::in | ios::binary);

    if(!in.is\_open()){

        cout << "Fail to open the encoded file!" << endl;

        return false;

    }

    ofstream out(source\_filename.c\_str(), ios::out | ios::binary);

    if(!out.is\_open()){

        cout << "Fail to open the output file!" << endl;

        return false;

    }

**size\_t**  sizes= 0;

    in.read((**char** \*)&sizes, 8);

*//首先从压缩文件的其实字符串得到文件的解码词典 之后便可根据解码词典进行解码*

**int** dictionary\_size = 0;

**unsigned** **char** temp;

    in.read((**char** \*)&temp, 1);*//解码字典的长度*

*//编码词典最多256个字符，由于unsigned char的表示范围为0~255，因此当解码词典个数为256时temp=0*

    dictionary\_size = temp ? (**int**)temp : 256;

    map<string, **char**> decode\_dictionary;

**char** ch;

**unsigned** **char** buf;

**unsigned** **char** len;

    string huffman\_code;

**unsigned** **char** buf\_index;

**unsigned** **char** i;

*//table数组 从LSB到MSB依次为1，其他位为0，用于往缓冲区buf逐位写入信息时的位选*

**unsigned** **char** table[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

    for (**int** j = 0; j < dictionary\_size;j++){

        buf = 0;

        buf\_index = 0;

        in.read((**char** \*)&ch, 1);*//读入解码字典的字符*

        in.read((**char** \*)&len, 1);*//读入当前字符编码的编码长度*

        in.read((**char** \*)&buf, 1);

        huffman\_code = "";

*//根据长度读出后续的编码01串*

        for ( i = 0; i < len; i++)

        {

            if(buf&table[buf\_index])

                huffman\_code += "1";

            else

                huffman\_code += "0";

            buf\_index++;

            if(buf\_index==8){

                buf\_index = 0;

*//若i=len-1表示刚好处理完缓冲区的字符，若没处理完则继续读入*

                if(i!=len-1)

                in.read((**char** \*)&buf, 1);

            }

        }

*//将获得的字符对应的编码插入解码器*

        decode\_dictionary.insert(map<string, **char**>::value\_type(huffman\_code, ch));

    }

*//根据获得的解码器对文件进行解码*

    map<string,**char**>::iterator it;

    buf\_index = 0;

    huffman\_code = "";

*//二进制读入压缩文件中的一个字符*

    in.read((**char** \*)&buf, 1);

*// 若文件没有结束则按照二进制一个字符一个字符的读入所有内容*

    while(!in.eof()){

        if(buf&table[buf\_index])

            huffman\_code += "1";

        else

            huffman\_code += "0";

        buf\_index++;

*//前缀编码中，根据编码找到对应的字符，将字符写入解压文件*

        it = decode\_dictionary.find(huffman\_code);

        if(it!=decode\_dictionary.end()){

            ch = it->second;

            out.write((**char** \*)&ch, 1);

            --sizes;

            if(sizes==0){

                in.close();

                in.clear();

                out.close();

                out.clear();

                return true;

            }

            huffman\_code = "";

        }

*//读入的字符处理完毕，继续读入一个字符*

        if(buf\_index==8){

            buf\_index = 0;

            in.read((**char** \*)&buf, 1);

        }

    }

}

压缩

1. 共打开两次输入文件，一次用于统计字符出现频率和文件长度，一次用于编码写入压缩文件
2. 若原始文件或压缩文件打卡失败时报错
3. 用table数组作为缓冲区的位选
4. 压缩后，文件中的数据存储顺序为：

原始文件长度sizes|解码词典元素个数n|第1个字符键值|第1个字符对应的Huffman编码长度|第1个字符的Huffman编码的存储区|第2个字符键值|第2个字符对应的Huffman编码长度|第2个字符的Huffman编码的存储区|···|第n个字符键值|第n个字符对应的Huffman编码长度|第n个字符的Huffman编码的存储区|原始文件的Huffman编码|原始文件的Huffman编码长度不是8的倍数时补0

解压

1. 打开压缩文件读取原始文件长度
2. 读取解码词典元素个数，再根据解码词典元素个数读取解码词典内容：依次读取字符键值、字符对应Huffman编码长度，再根据字符对应Huffman编码长度在字符的Huffman编码的存储区读取字符对应的Huffman编码，读取完毕时即构建好了解码词典
3. 根据解码词典对原始文件的Huffman编码进行译码，写入输出文件中。
4. 其中输出文件被保存成了指定的格式。
5. 解码完毕，实现无损解压。

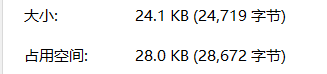
**实验结果**

下面展示压缩txt（文本）、jpg（图片）格式文件的压缩和解压，其他格式文件同理。

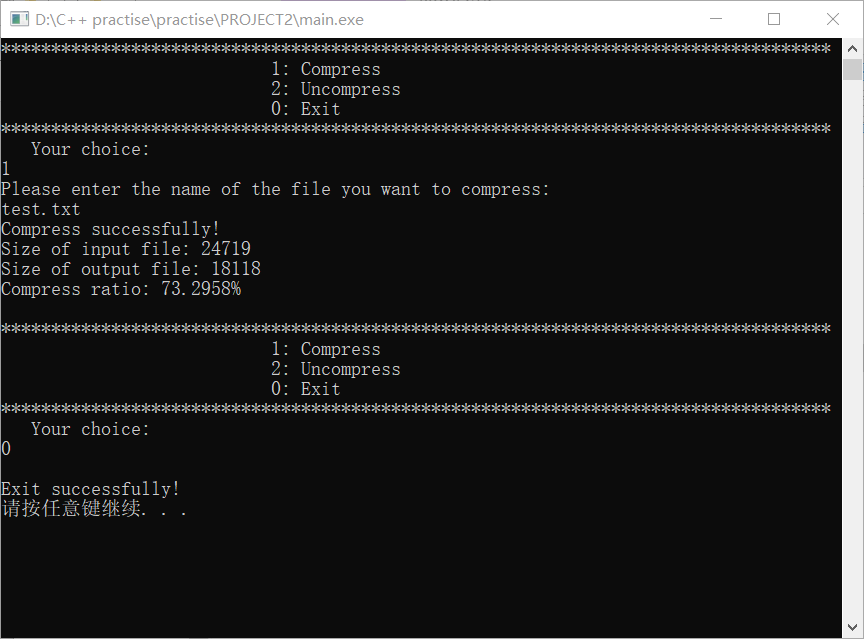
test.txt



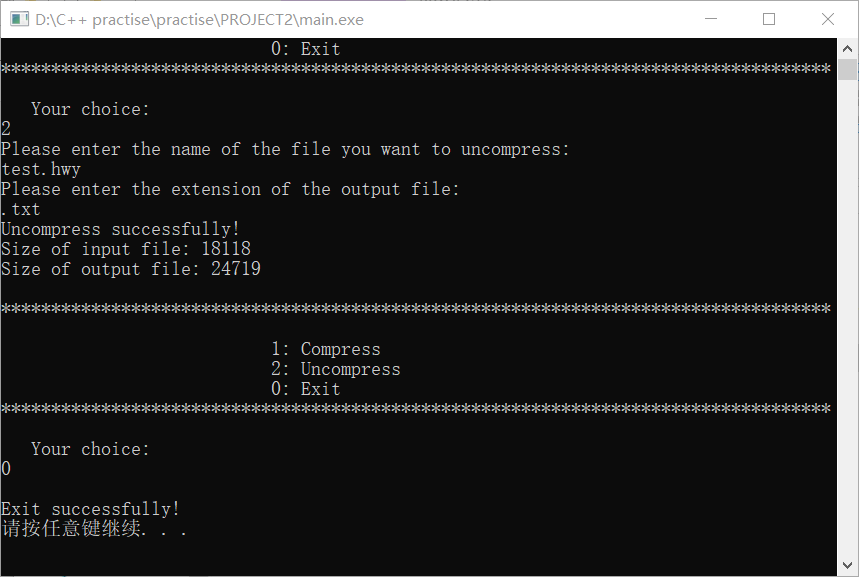
文件信息



压缩文件



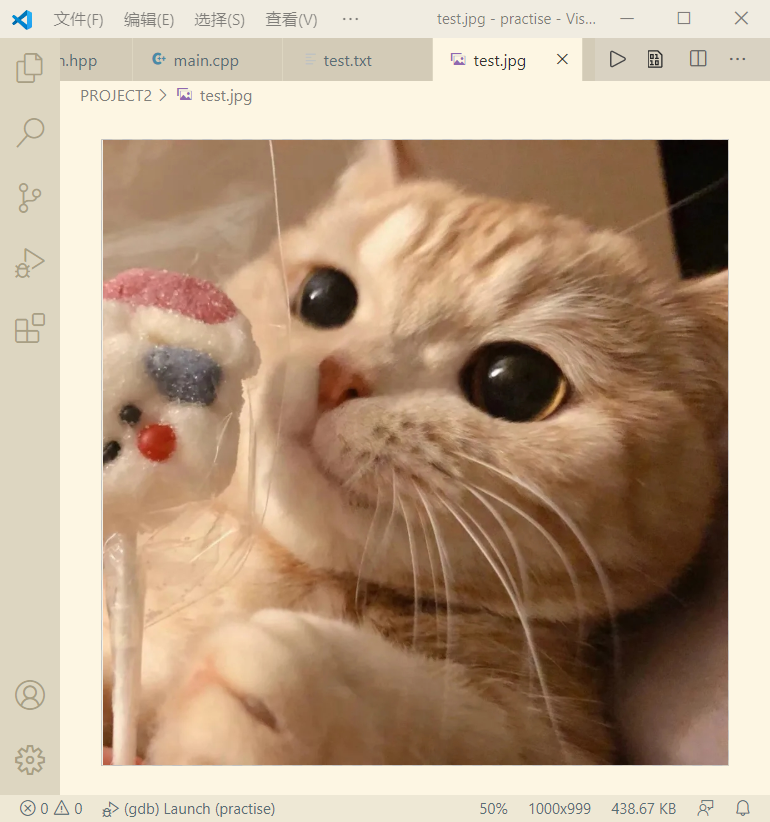
解压文件



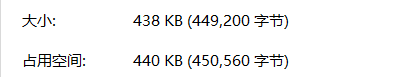
test.txt

解压后解压文件和原始文件大小相同，内容相同，符合预期效果。

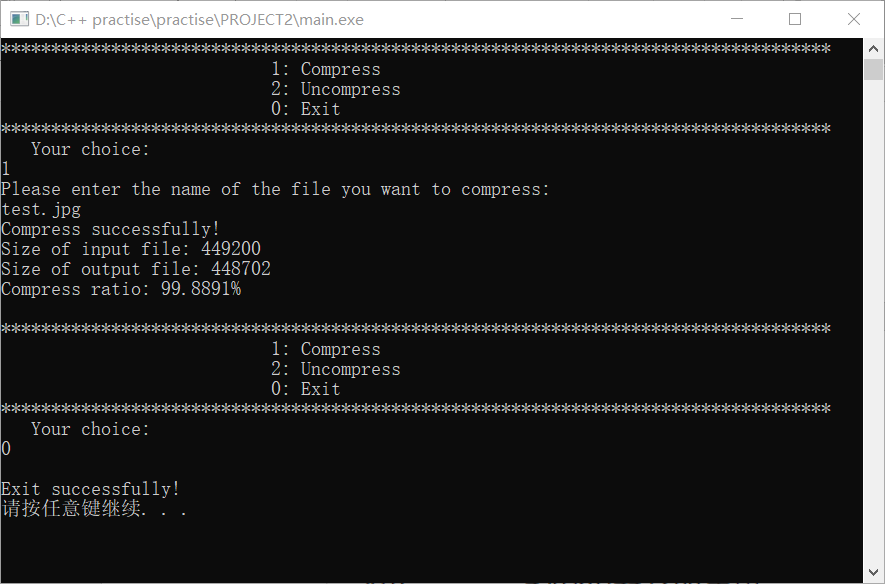
test.jpg



文件信息：

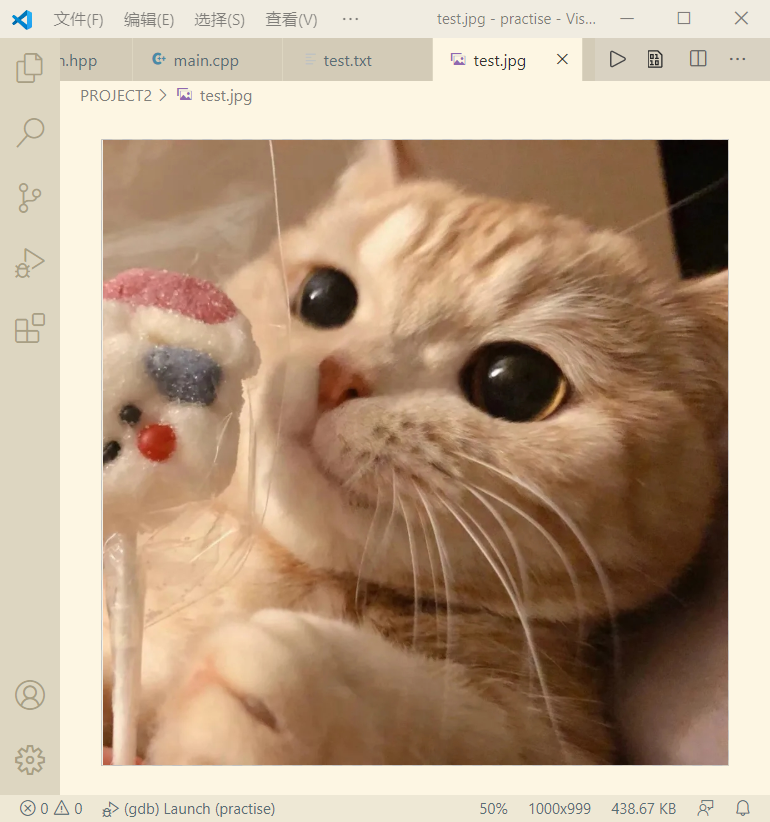


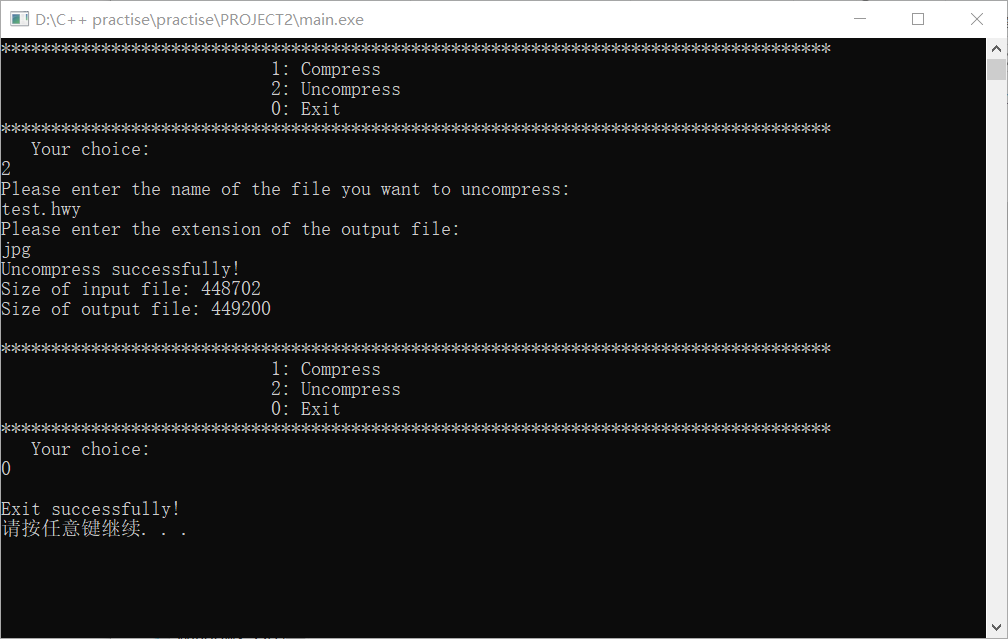
压缩



解压后

test.jpg





解压后解压文件和原始文件大小相同，内容相同，符合预期效果。

反思：

该程序在压缩字符出现频率较多且类型较少的文件时压缩效果较好，压缩比率较低，如以上压缩txt文件的例子，但在压缩字符种类较多的文件时压缩效果一般，如以上压缩jpg格式的文件的例子。

此外，当压缩较小的文件时压缩效果不佳，因为需要往压缩文件写入解压信息（原始文件长度、解码词典）需要占据一定的空间，当Huffman算法对该文件带来的优化效果不能抵消附加解压信息带来的负面效果时，压缩是没有意义的，甚至会出现压缩文件大于原始文件的情况，这些是我认为值得注意的问题。

五、**实验心得**

通过这次实验，我：

1. 理解和掌握了Huffman树和Huffman算法，明白了Huffman算法用于压缩文件的原理。
2. 学会了如何构造一颗Huffman树
3. 明白了何为前缀码，前缀码的特点和应用
4. 理解了文件在系统中的存储形式和文件拓展名的作用

此次实验使我受益匪浅。

参考资料

《哈夫曼树和哈夫曼编码》-乔海燕

<https://www.cnblogs.com/qionglouyuyu/p/4850679.html>

<https://blog.csdn.net/sugarbliss/article/details/80315327>