

# 高程实验报告

报告名称:基于GZIP中LZ77算法的压缩小程序

专业/班级:新生院济勤学堂八班

姓 名:姚忠豪

学 号:1952731

完成日期:2020/4/24

修改日期:2020/4/27

# 设计思路 ⚡

## 数据压缩原理

数据压缩 (Data Compress) 的原理是按照一定的算法对数据进行重新组织, 减少数据的冗余和存储的空间。

常见的无损数据压缩算法有LZW压缩, LZ77压缩, Huffman编码压缩等等。

LZW算法:

LZW算法中, 首先建立一个字符串表, 把每一个第一次出现的字符串放入串表中, 并用一个数字来表示, 这个数字与此字符串在串表中的位置有关, 并将这个数字存入压缩文件中, 如果这个字符串再次出现时, 即可用表示它的数字来代替, 并将这个数字存入文件中。压缩完成后将串表丢弃。如"print" 字符串, 如果在压缩时用266表示, 只要再次出现, 均用266表示, 并将"print"字符串存入串表中, 在图象解码时遇到数字266, 即可从串表中查出266所代表的字符串"print", 在解压缩时, 串表可以根据压缩数据重新生成。

经典LZ77算法:

LZ77算法就是把数据中一些可以组织成短语(最长字符)的字符加入字典, 然后再有相同字符出现采用标记来代替字典中的短语, 如此通过标记代替多数重复出现的方式以进行压缩。主要算法逻辑就是, 先通过前向缓冲区预读数据, 然后再向滑动窗口移入(滑动窗口有一定的长度), 不断的寻找能与字典中短语匹配的最长短语, 然后通过标记符标记, 标记包含三部分信息: (滑动窗口中的偏移量(从匹配开始的地方计算)、匹配中的符号个数、匹配结束后的前向缓冲区中的第一个符号)。若找不到匹配, 则将非匹配的字符编码为本身。

最后以标识符0, 1来区分是否 ' 编码为本身 ' 或者 ' 编码为了 < 距离,长度 ,字符 >对 '

GZIP中的LZ77算法:

GZIP的压缩算法主要包括改进的LZ77算法和Huffman算法两种压缩方法, 其中的LZ77算法是经典LZ77算法的改进版。

其查找匹配过程采用了哈希表的方式以提高查找匹配效率, 窗口大小选择的是64K (包括查找缓冲区和先行缓冲区各32K), 此外当查找匹配到最长串时, 编码到压缩文件的不是 < 距离,长度 ,字符 >对, 而只是 < 距离,长度 >对, 其中距离为2字节, 长度为1字节 (也就对应着最小的匹配长度为3)

Huffman算法：

Huffman算法是通过Huffman树将具有不同权值（通过该字符在文件中出现的频率决定）的字符用一串编码（0 1组成）代替，权值大的编码短，权值小的编码长，于是就达到了去除冗余以减小储存空间的目的。

其中构造Huffman树的过程利用了哈希桶和二叉树，最后将所有字符都以“位”的形式替换到压缩文件中，解压文件的过程就是从压缩文件中一位一位地读取数据然后通过Huffman树将字符还原出来。

## 选择压缩算法

当对算法的选择无从下手时，我选择了先观察大作业给出的目标日志文件（ser.log），通过文件的特点选择相应的压缩算法。毕竟有一些算法还是比较难实现的，就算实现了，要真正理解起来还是有一定困难的。

下面是ser.log文件的结构：

```
_config.yml — myblog X serlog X demo.cpp X _config.yml — MyBlog(themes)Butterfly X
1 #Software: Hllpoj Server 1.0.1 / Logger 1.0.0 built 0001
2 #Version: 1.0
3 #Date: 2019-03-13 00:00:00
4 #Fields: date time s-ip cs-method cs-uri-stem cs-uri-query s-port cs-username c-ip cs(User-Agent) cs(Referer) sc-status sc-substatus
5 sc-win32-status time-taken
6 2019-03-13 00:00:00 ***.***.***.*** POST /login.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2
http://***.***.***.***index.html 200 0 0 114
7 2019-03-13 00:00:00 ***.***.***.*** POST /default/*****.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2
http://***.***.***.***default/*****.htm 200 0 0 23
8 2019-03-13 00:00:00 ***.***.***.*** POST /default/*****.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2
http://***.***.***.***default/*****.htm 200 0 0 2
9 2019-03-13 00:00:00 ***.***.***.*** POST /default/*****.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2
http://***.***.***.***default/*****.htm 200 0 0 3
10 2019-03-13 00:00:06 ***.***.***.*** POST /default/*****.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2
http://***.***.***.***default/*****.htm 200 0 0 41
11 2019-03-13 00:00:06 ***.***.***.*** POST /default/*****.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2
http://***.***.***.***default/*****.htm 200 0 0 3
12 2019-03-13 00:00:06 ***.***.***.*** POST /default/*****.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2
http://***.***.***.***default/*****.htm 200 0 0 4
13 2019-03-13 00:00:08 ***.***.***.*** POST /default/*****.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2
http://***.***.***.***default/*****.htm 200 0 0 53
14 2019-03-13 00:00:08 ***.***.***.*** POST /default/*****.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/
```

可见，本日志文件除了大之外，另一个特点就是有大量重复的字符串（这是由于日志文件本身的格式决定的）

日志文件格式：

1. 2019-03-13: 是用户访问时间。
2. POST: 服务器的处理动作，包括GET和POST。网站日志中绝大部分都是GET，只有在进行CGI处理的时候才会出现POST。GET，就是用户从服务器上获取了页面或者别的文件。
3. POST后面“/...”: 是用户访问的页面，只有一个斜杠表示网站首页
4. &&&&&&&&&: 用户IP地址。通过用户IP，可以查询到用户来自哪个国家、省份、城市。
5. Mozilla/5.0+(Linux+.....): 表示用户所使用的电脑是Mozilla浏览器，操作系统等等
6. .... 200 360: 这是代码中最重要的信息。前面是用户访问自己网站的某一个页面，后面的200，表示用户访问页面的时候返回的状态码。200后面的360代表的是被访问页面的体积。

由此可以看出此文件包含着大量重复的冗余的字符串信息。那么参考之前提到的数据压缩的方法，最好的选择便是基于字典的压缩算法，而不选择Huffman编码算法，因为Huffman编码与基于字典的算法在维度上就有不同，前者侧重的是全局的字符层面的冗余，而后者侧重的是局部语句层面的冗余。当然在能力允许的范围内最好是结合这两种算法（比如高效的压缩方法GZIP），但是出于实现的难度，我最终选择了GZIP中的LZ77算法作为主要压缩方法。

# LZ77算法具体流程

了解哈希表：

哈希表是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。在这里，哈希表的key值就是三字符的字符串，而得到的位置就是该字符串在当前窗口中的相对位置。

本算法中的哈希表head数组的大小为32K，为了解决哈希冲突的问题（即当新出现的字符串的哈希地址与之前的字符串相同而导致值被覆盖，从而影响匹配的效率），引进了一个prev数组（32K），用此数组与head数组形成一种链式结构的匹配链，以此来提高查找匹配效率

## 压缩过程

- 打开待压缩文件（二进制打开）
- 读取一个窗口的数据（64K）
- 通过开始两个字符初始化哈希地址，之后每读一个字符就可以将此字符和其前面两个字符组成的字符串的哈希地址计算出来
- 循环开始压缩
  - 计算哈希地址，将该字符串首字符在窗口中的位置插入到哈希表中，并返回该表的状态matchHead
  - 根据matchHead检测是否找到匹配
    - matchHead等于0，表示未找到匹配，该三个字符在前文中没有出现过，将该当前字符作为源字符写到压缩文件
    - matchHead不等于0，表示找到匹配，matchHead代表匹配链的首地址，从哈希桶matchHead位置开始找最长匹配，找到后用该<长度,距离>对替换该字符串写到压缩文件中，然后将该替换串三个字符一组添加到哈希表中
- 如果窗口中的数据小于MIN\_LOOKAHEAD时，将右窗口中数据搬移到左窗口，从文件中新读取一个窗口的数据放置到右窗，更新哈希表，继续压缩，直到压缩结束

## 压缩数据保存

压缩数据分为两个文件保存

- 压缩数据，用以保存压缩后的数据或者<长度,距离>数据
- 标记信息，用以保存标记当前字节是源字符还是<长度,距离>对

## 解压缩过程

- 从标记文件中读取标记
- 如果当前标记是0，表示源字符，从压缩数据文件中读取一个字节，直接写到解压缩之后的文件中
- 如果当前标记是1，表示遇到<长度,距离>，从压缩数据文件中先读取一个字节作为压缩的长度，再继续读取两个字节作为距离，然后从解压缩过的结果中找出匹配长度
- 获取下一个标记，直到所有的标记读取完

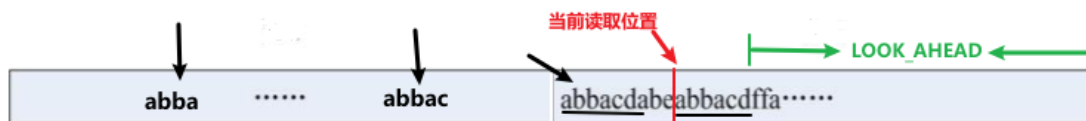
## 图解

1-1 滑动窗口示意图



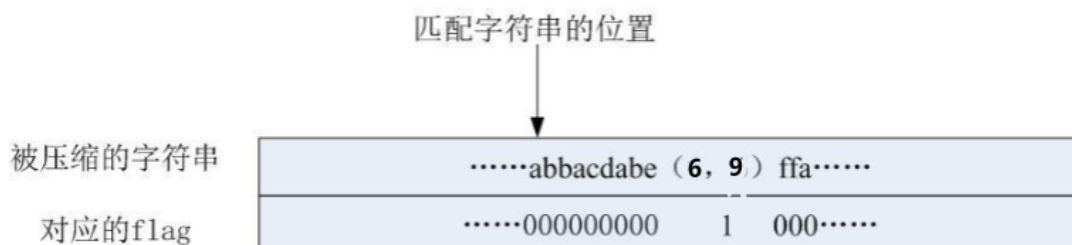
每次读取一个字符，通过该字符和哈希函数计算出以该字符结尾的三字符字符串的哈希地址s，若head[s]为空，则代表前面没有该字符串的匹配串，若head[s]不为空，则说明之前至少有一个匹配串，于是就要查找最长匹配

1-2 查找最长匹配



用以上的例子说明，通过计算"abb"的哈希地址，发现之前有匹配串，并且得到了一个匹配链，再通过该匹配链找出最长匹配串，如图，最长的匹配链应该为"abbacd"，于是就将当前字符串"abbacd"替换为<长度，距离>=<6,9>写入到压缩文件中，并且向标记文件中添加标记位'1'，再将"abbacd"三个字符为一组插入到哈希表中。当前读取位置移动到'f'处，当LOOK\_AHEAD的大小小于MIN\_LOOKAHEAD时，将窗口滑动32K

1-3 解压缩过程



解压缩过程就是通过标志位来确定是否要读取为<长度，距离>对，如图，当标志位为'1'时，读取<6,9>，将距离'e'9个字符处开始，长度为6的字符串"abbacd"写入解压后文件

## LZ77算法实现

### 哈希表类的实现

**哈希表**是GZIP中LZ77算法提高效率的关键，合理地选择哈希表的大小和匹配最大匹配次数对于匹配效率有很大的影响。

哈希表的主要功能就是保存之前出现过的字符串（长度为3）在窗口中的相对位置以及将同样的字符串（长度为3）形成匹配链以确保找到最长匹配长度的串。其中最主要的函数为哈希函数与插入函数：

```

/*****
功能: 获得字符串的哈希函数
参数: hashAdr 上一个串的哈希地址 ch 要计算哈希地址的字符串
      第三个字符
    
```

返回值: void

描述: 通过传入字符串（三字符）的第三个字符来获取本字符串的哈希地址，（计算字符串哈希地址需要三个字符，只传入一个字符的原因是前两个字符的哈希地址已经被保存在上一个字符串的哈希地址中，所以只需要第三个字符就可以将该串的哈希地址计算出

```
*****/
void HushTable::HushFunction(unsigned short& hashAddr, unsigned char ch) {
    hashAddr = (((hashAddr) << HUSH_SHIFT()) ^ (ch)) & HUSHMASK;
}

unsigned short HushTable::HUSH_SHIFT() {
    return (HASH_BITS + MIN_MATCHLENGTH - 1) / MIN_MATCHLENGTH;
}
//HUSHMASK=32*1024-1
//HASH_BITS=15
//MIN_MATCHLENGTH=3
```

/\*\*\*\*\*\*

功能: 通过字符串的哈希地址将其位置插入到哈希表中

参数: matchHead 匹配头 ch 要插入的字符串的第三个字符 pos 当前串的位置 hashAddr 上一个串的哈希地址

返回值: void

描述: 每从先行缓冲区读取一个字符时，就计算其与其后两个字符组成的字符串的哈希地址（head数组的索引），获得对应head数组元素（若为0说明无匹配，即之前没有出现过该串），再将该元素赋给prev数组中索引为当前位置的元素，从而形成匹配链

```
*****/
void HushTable::HushInsert(unsigned short& matchHead, unsigned char ch, unsigned short pos, unsigned short& hashAddr) {
    //通过哈希函数计算出当前串的哈希地址
    HushFunction(hashAddr, ch);

    //传入匹配链头（用于查找匹配串），再将head[hashAddr]的值（离本串最近的匹配串的位置）赋给匹配链头
    matchHead = Head[hashAddr];

    //将head[hashAddr]的值（离本串最近的匹配串的位置）赋给prev数组中索引为当前位置的元素，从而形成匹配链
    Prev[pos & HUSHMASK] = Head[hashAddr];

    //将当前位置赋给匹配链的头
    Head[hashAddr] = pos;
}
```

HUSHMASK详解:

查找缓冲区最多有32K，数组Prev[]共有32K个元素，而该数组索引的实际意义又是字符串的位置，所以我们将该数组与查找缓冲区联系起来。我们知道，原始数据是放到Window中的，Window大小为64K；先行缓冲区的第一个字节strStart随着压缩的逐字节进行而在Window中不断推进，势必要大于32K（strStart的范围是闭区间[0, 64\*1024- MIN\_LOOKAHEAD]，前面提到过一点），而strStart的位置其实就是当前字符串的位置。prev[]只有32K个元素，其索引就是字符串的位置，而strStart如果大于32K，那还怎么把字符串插到字典中？prev[strStart]就数组越界了，为了解决这个问题，压缩算法内部对prev[]的索引使用了一个非常巧妙而又简单易懂的处理方法：让strStart与HUSHMASK按位与运算的结果作prev[]的索引，而不是直接用strStart作prev[]的索引，其中，HUSHMASK的值是32768，即prev[strStart & HUSHMASK]。这样就保证了在

strStart的取值范围内，prev[]的索引与strStart都是完全对应的。例如，strStart = 300，那么strStart & HUSHMASK = 300；strStart = 32968 = 32K + 200，那么strStart & HUSHMASK = 200。这么看来，prev[]数组可以做到在strStart超过32K的时候，始终保证匹配链的长度在32K的范围内，用新的字符串位置值替换掉同一索引下旧的字符串位置值（旧的位置与当前字符串的位置之间的距离有可能已经超过32K，就算不超过也是所有匹配串中最远的）

此外还有需要实现的函数有“更新哈希表”（在每一次窗口滑动之后都要更新哈希表），“获得下一个匹配串位置”（查找匹配过程中，通过Prev数组的匹配链不断获取匹配串位置）

## 核心压缩函数实现

```
/******  
功能： 核心压缩函数，将文件通过LZ77算法压缩  
参数： fileName 文件名 newFileName 压缩后的文件名  
返回值： void  
描述： 通过滑动窗口中的先行缓冲区读取的字符（和其后两个字符构成  
        的字符串）建立哈希表，同时建立匹配链，再用当前串通过哈希  
        表的匹配链查找最长匹配，将<长度，距离>对替换该串（当找不  
        到匹配就不做替换，继续读入下一字符）；再将每个位置对应的  
        标记信息（是否被‘<长度，距离>对’替换了）写入到标记文件  
*****/  
void LZ77::CompressFile(string fileName, string newFileName) {  
  
    //打开要压缩的数据文件  
    ifstream fIn(fileName, ios::in | ios::binary);  
    if (!fIn.is_open()) {  
        cout << "open file error!";  
        exit(0);  
    }  
  
    //打开压缩后的数据文件  
    ofstream fOut(newFileName, ios::out | ios::binary);  
    if (!fOut.is_open()) {  
        cout << "open file error!";  
        exit(0);  
    }  
  
    //打开临时的标记信息文件  
    ofstream fFlag("flag.txt", ios::out | ios::binary);  
    if (!fFlag.is_open()) {  
        cout << "open file error!";  
        exit(0);  
    }  
  
    //获取文件大小  
    fIn.seekg(0, ios::end);  
    unsigned long long fileSize = fIn.tellg();  
    fIn.clear();  
    fIn.seekg(0, ios::beg);  
  
    //先从文件中读取64K（32K+32K）个字节，并且获得当前读取的字节数作为先前缓冲区  
    fIn.read((char*)window, HWS * 2);  
    size_t lookAhead = size_t(fIn.gcount());  
  
    //首先利用前两个字符设置最初的哈希地址
```



```

unsigned short hashAddress = 0;
LZhush.HushFunction(hashAddress, window[0]);
LZhush.HushFunction(hashAddress, window[1]);

//与查找匹配相关的变量
unsigned short strStart = 0;           //当前匹配查找的位置（即先前缓冲区的第一个
字符在窗口中的相对位置）
unsigned short matchHead = 0;          //当前字符串匹配查找的头位置，若为0则没有找
到匹配串（即离当前串距离最近的匹配串的位置）
unsigned char curMatchLength = 0;      //当前串的匹配长度
unsigned short curMatchDistance = 0;   //当前匹配的距离（当前串与匹配串之间的距
离）

//与写入标记信息相关的变量
unsigned char chFlag = 0;              //8bit标记信息
unsigned char bitCount = 0;           //标记信息bit位

//开始查找匹配
while (lookAhead) {

    //插入三字符字符串中的最后一个字符以计算哈希地址，获取匹配头，完善匹配链
    LZhush.HushInsert(matchHead, window[strStart + 2], strStart,
hashAddress);

    //更新与当前匹配相关的变量
    curMatchLength = 0;                //当前的匹配长度
    curMatchDistance = 0;              //当前匹配的距离（当前串与匹配串之间的距离）

    //当获取的匹配头不为0，则通过GetLonggestMatchLength函数得到匹配链中的最长匹配长度
    if (matchHead) {
        curMatchLength = GetMaxMatchLength(matchHead, curMatchDistance,
strStart);
    }

    //验证是否真的在匹配链中找到了不小于MIN_MACHLENGTH（最短的匹配长度）的匹配串
    if (curMatchLength >= MIN_MATCHLENGTH) {

        //说明找到了有效匹配串，将<长度，距离>对写入压缩数据文件中
        unsigned char length = curMatchLength - 3;           //<长度>
        unsigned short distance = curMatchDistance;          //<距离>
        fout.put(length);                                     //写入长度
        fout.write((char*)&distance, sizeof(distance));      //写入距离

        //写入标记文件
        writeFlag(fFlag, chFlag, bitCount, true);

        //替换后将被替换的串中所有三字符字符串插入到哈希表中
        for (unsigned char i = 0; i < curMatchLength - 1; i++) {
            strStart++;           //后移一个字符
            LZhush.HushInsert(matchHead, window[strStart + 2], strStart,
hashAddress); //插入三字符字符串
        }
        strStart++;              //循环过程中少移了一次

        //更新先行缓冲区中剩余字符的个数
        lookAhead = lookAhead - curMatchLength;
    }
}

```



```

    }
    else {

        //说明没有找到有效匹配串，将字符原样输出到压缩数据文件中
        fOut.put(window[strStart]); //写入字符
        strStart++;                //后移一个字符
        lookAhead--;               //先前缓冲区减1

        //写入标记文件
        writeFlag(fFlag, chFlag, bitCount, false);

    }

    //当先行缓冲区的长度小于最小长度时，窗口滑动
    if (lookAhead <= MIN_LOOKAHEAD) {
        Movewindow(fIn, lookAhead, strStart);
    }
}

//当最后的标记位数不够8bit
if (bitCount > 0 && bitCount < 8) {

    //直接写入标记文件
    chFlag <= (8 - bitCount);
    fFlag.put(chFlag);
}

//关闭标记文件
fFlag.close();

//合并压缩数据文件与标记文件形成最终压缩文件
MergeFile(fOut, fileSize);

//关闭各文件
fIn.close();
fOut.close();

// 将用来保存标记信息的临时文件删除掉
remove("flag.txt");
}

```

其中LZhush为LZ77类的一个成员，类型为哈希表类（HushTable），用于存放窗口Window对于的哈希表

MoveWindow函数为滑动窗口函数，包括更新窗口内的数据和更新其对应的哈希表：

```

/*****
功能： 滑动窗口
参数： fIn 要压缩文件 lookAhead 先前缓冲区剩余字节数 strStart
      当前串所在位置（即先前缓冲区的第一个字符位置）
返回值： void
描述： 先将右窗中的数据移至左窗，再向右窗补充数据，最后别忘了更新哈希表
*****/
void LZ77::Movewindow(istream& fIn, size_t& lookAhead, unsigned short&
strStart) {

    //先判断是否到达了文件末端

```

```

if (strStart >= HWS) {

    //将右窗中的数据移至左窗
    memcpy(window, window + HWS, HWS);

    //将右窗置0
    memset(window + HWS, 0, HWS);
    strStart -= HWS;

    //更新哈希表
    LZhash.UpdateHash();

    //向右窗（先前缓冲区）补充数据
    if (!fIn.eof()) {
        fIn.read((char*)window + HWS, HWS);
        lookAhead = lookAhead + size_t(fIn.gcount());
    }
}
}
//其中HWS大小为32K

```

最后得到两个文件（压缩数据文件与对应的标记文件）将两文件合并，再删除临时的标记文件就得到了最终的压缩文件。

## 解压函数的实现（待优化）

解压过程实则是最容易理解，也是操作方便的，但是我在写解压函数的过程中总是遇到各种问题，（在路遇问题部分会提到），最终勉强能写出一个解压函数，但是解压时间感人（压缩的效率却快的一批 😊），所以解压函数还有很大的优化空间。

```

/*****
功能： 解压函数，将文件解压
参数： fileName 文件名  newFileName 解压后的文件名
返回值： void
描述： 从标记文件中读取标记，如果当前标记是0，表示源字符，从压缩数据文件中
        读取一个字节，直接写到解压之后的文件中；如果当前标记是1，表示
        遇到<长度,距离>，从压缩数据文件中先读取一个字节作为压缩的长度，再
        继续读取两个字节作为距离，然后从解压过的结果中找出匹配长度
*****/
void LZ77::UnCompressFile(const string fileName,const string newFileName)
{
    // 操作压缩文件 "原始数据" 的指针
    FILE* fIn = fopen(fileName.c_str(), "rb");
    if (nullptr == fIn)
    {
        std::cout << "压缩文件打开失败" << std::endl;
        return;
    }

    // 操作压缩文件 "标记数据" 的指针
    FILE* fInFlag = fopen(fileName.c_str(), "rb");
    if (nullptr == fInFlag)
    {
        std::cout << "压缩文件打开失败" << std::endl;
        return;
    }

```

```

}

// 获取原始文件的大小
unsigned long long fileSize = 0;
fseek(fInFlag, 0 - int(sizeof(fileSize)), SEEK_END);
fread(&fileSize, sizeof(fileSize), 1, fInFlag);

// 获取标记信息的大小
unsigned long long flagSize = 0;
fseek(fInFlag, 0 - int(sizeof(fileSize)) - int(sizeof(size_t)), SEEK_END);
fread(&flagSize, sizeof(fileSize), 1, fInFlag);

// 将标记信息文件的文件指针移动到保存标记数据的起始位置
fseek(fInFlag, 0 - int(sizeof(fileSize)) - int(sizeof(size_t)) -
long(flagSize), SEEK_END);

//打开解压文件
FILE* fOUT = fopen(newFileName.c_str(), "wb");

//用于从已经解压好的数据中向前读取重复的字节
FILE* fOutIn = fopen(newFileName.c_str(), "rb");

unsigned char bitCount = 0;
unsigned char chFlag = 0;
unsigned long long encodeCount = 0;    //已经解码的长度

while (encodeCount < fileSize){

    //处理完完8bit时，再读取8bit
    if (0 == bitCount)
    {
        chFlag = fgetc(fInFlag);
        bitCount = 8;
    }

    if (chFlag & 0x80)
    {
        // 距离长度对
        unsigned short matchLength = fgetc(fIN) + 3;
        unsigned short matchDistance = 0;
        fread(&matchDistance, sizeof(matchDistance), 1, fIN);

        // 清空缓冲区：系统会将缓冲区中的数据写入到文件中
        fflush(fOUT);

        // 更新已经解码字节数大小
        encodeCount += matchLength;

        //移动文件指针至匹配字符处
        fseek(fOutIn, 0 - matchDistance, SEEK_END);

        //写入匹配的字符
        for (size_t i = 0; i < matchLength; i++) {
            fputc((fflush(fOUT), fgetc(fOutIn)), fOUT);
        }
    }
}

```

```

else
{
    //写入原字符
    int ch = fgetc(fIN);
    fputc(ch, fOUT);
    encodeCount++;
}




//再读取一个bit位
chFlag <=<= 1;
bitCount--;
}

//关闭相关文件
fclose(fIN);
fclose(fInFlag);
fclose(fOUT);
fclose(fOutIn);
}

```

最后得到的解压缩文件通过对比是一致的(ser.log 原文件 ser.tj 压缩文件 ser.tj.log 解压后的文件)。

2-1










 ser.log	2020/4/8 21:25	文本文档	10,073 KB
 ser.tj	2020/4/16 13:09	TJ 文件	475 KB
 ser.tj.log	2020/4/16 13:09	文本文档	10,073 KB



## 功能描述

### 对不同文件的压缩

文件类型	文件大小 / 重复度	压缩时间	压缩比(压缩文件大小/原文件大小)	解压时间 (待优化)
文本文件	10073KB / 较高	1.625 s	4.7%	27.234 s
文本文件	4925KB / 低	2.078 s	97%	12.187 s
png图片文件	650KB / ~	0.25 s	84%	1.109 s

 1010.tj	2020/4/26 14:55	TJ 文件	4,789 KB
 1010.tj.txt	2020/4/26 14:58	TXT 文件	4,925 KB
 1010.txt	2020/4/15 19:40	TXT 文件	4,925 KB
 1012.png	2017/4/29 5:45	PNG 文件	650 KB
 1012.tj	2020/4/26 15:00	TJ 文件	551 KB
 1012.tj.png	2020/4/26 15:01	PNG 文件	650 KB
 ser.log	2020/4/8 21:25	文本文档	10,073 KB
 ser.tj	2020/4/26 14:56	TJ 文件	475 KB
 ser.tj.log	2020/4/26 14:58	文本文档	10,073 KB

## 压缩效率分析

对于一般的文本文件（文本重复度适中），可以达到50%的压缩比，但是当文本文件的重复度极低时，压缩效率就很不乐观，有时甚至出现压缩文件大于原文件的情况，而对于图片文件，压缩效率普遍不怎么理想。由此分析GZIP中的LZ77算法对文件的重复程度依赖性很高，如果再结合Huffman算法压缩，其压缩就会更加高效。

## 路遇问题与解决办法 ☹

### 查找匹配串进入死循环 ✓

查找最长匹配的过程原理：

当获得的Head[hashAddress]不为空时(即不为0)时，我们需要返回这个链式结构的头（匹配头），然后再沿着这个链式结构寻找最长的匹配链，直到该链某节点的数值为空（即为0）时，停止匹配，获得最长匹配链对于的<长度，距离>对。

- 问题：之前我们介绍HUSHMASK的时候说到过，当使用新的位置覆盖prev[strStart&HUSHMASK]的时候，这个被覆盖的值有可能是0，即“再往前没有匹配串”。如果没被覆盖，搜索匹配链时如果到了这个节点，就不会再继续再搜了；但是现在被覆盖了，也就是说表示“再往前没有匹配串”这个语义的节点不存在了，那查找就会陷入死循环。
- 解决办法：通过设置一个最大匹配次数，记为maxMatchTimes，每次沿着匹配链查找的时候，最多找maxMatchTimes次。这样不仅能够解决这种潜在的死循环问题，还可以通过设置maxMatchTimes为不同的值来掌控压缩的效率。
- 下面贴上查找最长匹配的函数实现：

```

/*****
功能： 获得最长的匹配串
参数： matchHead 匹配头 strStart 当前字符串的位置 curcurMat-
chDistance 当前串与匹配串的距离
返回值： 最长的匹配长度
描述： 通过匹配链不断查找得到最长匹配长度的匹配串，但是要通过限制匹配次数（次数可自定义，这影响了压缩的程度和时间）以防止陷入死循环
*****/
unsigned char LZ77::GetMaxMatchLength(unsigned short matchHead, unsigned short& curMatchDistance, unsigned short strStart) {

```

```

//相关的变量
unsigned char nowMatchLength = 0;    //每次匹配的长度
unsigned short curMatchStart = 0;    // 当前匹配在查找缓冲区中的起始位置
unsigned char maxMatchTimes = 255;  //最大的匹配次数
unsigned char maxMatchLenght = 0;    //最长的匹配长度

do {
    //更新 从当前串开始的匹配范围（先行缓冲区中）
    unsigned char* pStart = window + strStart;    //开始
    unsigned char* pEnd = pStart + MAX_MATCHLENGTH; //结束

    //更新 查找缓冲区的起始
    unsigned char* pMatchStart = window + matchHead;

    //更新 每次匹配的长度
    nowMatchLength = 0;

    //逐字符进行匹配
    while (pStart < pEnd && *pStart == *pMatchStart) {

        //使两串再后移一位进行匹配，再使匹配长度加1
        pStart++;
        pMatchStart++;
        nowMatchLength++;
    }

    //一次匹配结束了，判断并记录最长匹配
    if (nowMatchLength > maxMatchLenght) {
        maxMatchLenght = nowMatchLength;    //更新 长度
        curMatchStart = matchHead;          //更新 距离
    }

    //当匹配头为空（即为0）时或者达到了最大的匹配次数时，停止匹配
} while ((matchHead = LZHush.GetNextAdress(matchHead)) > 0 && maxMatchTimes--);

//获得最终最长匹配对应的距离
curMatchDistance = strStart - curMatchStart;

//返回最长匹配长度
return maxMatchLenght;
}

```

## 解压缩效率不高 ✕

- 解压过程很好理解，从标记文件中读取标记，为0则直接原字符输入，为1则读取一个<长度l，距离d>对，**把距离当前位置之前d处的长度为l的字符串写入文件中**。由于大作业要求使用fstream处理文件，所以我先使用了fstream对同一个文件进行读写，但是效果并不乐观（特别是当标记为1时，获取<长度,距离>对的过程是正确的，但是写入文件一直出问题），大多情况下都输出乱码或者二进制文件（如下图）：

```

_config.yml — myblog X ser.log X demo.cpp X ser.tj.log X _config.yml — MyBlog\themes\Butterfly X
#Software: Hllpoj Server 1.0.1 / Logger 1.0.0 built 0001
#Version: 1.0
#Date: 2019-03-13 00:00:00
#Fields: date time s-ip cs-method cs-uri-stem cs-uri-query s-port cs-username c-ip cs(User-Agent) cs(Referer)
sc-win32-status time-taken
2019-03-13 00:00:00 ***.***.0000000 POST /login.php - 80 - ***.***.0000000 Mozilla/5.0+(Linux;+U;+Android;+8
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537
http://***.***.0000000/index.html 200 0 00114
2019-03-13 00:00:00 ***.***.0000000 POST /default/***0000php - 80 - ***.***.0000000 Mozilla/5.0+(Linux;+U;+
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537
http://***.***.0000000/default/***0000htm 200 0 0023
2019-03-13 00:00:00 ***.***.0000000 POST /default/***0000php - 80 - ***.***.0000000 Mozilla/5.0+(Linux;+U;+
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537
http://***.***.0000000/default/***0000htm 200 0 002
2019-03-13 00:00:00 ***.***.0000000 POST /default/***0000php - 80 - ***.***.0000000 Mozilla/5.0+(Linux;+U;+
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537
http://***.***.0000000/default/***0000htm 200 0 003
2019-03-13 00:00:00 ***.***.0000000 POST /default/***0000php - 80 - ***.***.0000000 Mozilla/5.0+(Linux;+U;+
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537
http://***.***.0000000/default/***0000htm 200 0 0041
2019-03-13 00:00:00 ***.***.0000000 POST /default/***0000php - 80 - ***.***.0000000 Mozilla/5.0+(Linux;+U;+
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537
http://***.***.0000000/default/***0000htm 200 0 003
2019-03-13 00:00:00 ***.***.0000000 POST /default/***0000php - 80 - ***.***.0000000 Mozilla/5.0+(Linux;+U;+
OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537
http://***.***.0000000/default/***0000htm 200 0 004
2019-03-13 00:00:00 ***.***.0000000 POST /default/***0000php - 80 - ***.***.0000000 Mozilla/5.0+(Linux;+U;+

```

```

_config.yml — myblog X ser.log X demo.cpp X ser.tj.log X _config.yml — MyBlog\themes\Butterfly X
1 2353 6f66 7477 6172 653a 2048 6c6c 706f
2 6a20 5365 7276 6572 2031 2e30 2e31 202f
3 204c 6f67 6700 0000 0000 0000 3020 6275
4 696c 7420 3030 3031 0d0a 2356 6572 7369
5 6f6e 3a6a 2053 650d 0a23 4461 7477 6172
6 3230 3139 2d30 332d 3133 0000 003a 0000
7 0065 000d 0a23 4669 656c 6473 3a20 643a
8 6a20 2074 696d 6520 732d 6970 2063 732d
9 6d65 7468 6f64 656c 6473 7572 692d 7374
10 656d 732d 6d65 7468 6f64 7175 6572 7920
11 7469 706f 7274 7374 656d 7373 6572 6e61
12 6a20 2063 732d 6970 2063 2855 6d73 732d
13 4167 656e 7429 6a20 2063 5265 6665 7265
14 7229 2073 6374 686f 6174 7573 7265 7229
15 2075 6263 7468 6f61 7475 7372 6577 696e
16 3332 7229 2073 6374 686f 7469 6d65 2d74
17 616b 656e 0d0a 0a23 4461 7477 6172 3230
18 3139 2d30 332d 3133 0020 2a2a 2a2e 2d31
19 3300 7477 6172 3230 3120 504f 5354 202f
20 6c6f 6769 6e2e 7068 7020 2d20 3830 6172
21 322a 2a2a 2e2d 3133 0074 7761 7232 3031
22 204d 6f7a 696c 6c61 2f35 2e30 2b28 4c69
23 6e75 783b 2b55 3b2b 416e 6472 6f69 642b
24 383a 2048 6c3b 2b65 6e74 7374 3b2b 4d49
25 582b 322b 4200 0000 642f 4f50 5231 2e31
26 3730 3632 332e 3032 3729 2b41 7070 6c65
27 5765 624b 6974 2f35 3337 2e33 362b 284b
28 4854 4d4c 2c2b 6c69 6b65 2b47 6563 6b6f
29 292b 536f 6674 7761 722f 342f 352e 4368

```

- 不知道是文件流指针的原因还是某某原因，让我一直很困扰，debug了很久☹
- 最后勉强使用了c语言方式的FILE\*的方式，每次读取一个字符，然后清空缓冲区再写入（如前文 UnCompressFile函数）

解压缩的时间的确比较长，所以还需要优化，之前有许多思路，但是都以乱码告终，可能是自己脑子瓦特了☹

## 心得体会 ♥

## 关于算法



- 通过这次大作业，让我对算法这东西有了一点初步的了解，因为现在还没有上过数据结构有关的课，没有深入接触过算法，所以这次作业对我来说还是有很大的挑战性的🐣。
- 有一些算法是真的需要很长的时间（对我来说）才可以理解透的，理解了之后还要处理细节部分，最终还要达到可以运用在实际程序中的水平😏。
- 算法学习真的是核心，语言学习是基础，“语言是不同的招式，而算法是内功”，有一说一，就算把c++撸完也不可能徒手把这个大作业写出来，一定要借助参考文献和相关算法流程📖。

## 关于程序

- 我们日常使用的压缩程序（7-zip, zip, WinRAR...）实际上运用了如此高阶的算法，而且大多都是许多算法结合，优势互补，达到最大优化🔗。
- 肝一个程序的过程是及其艰难的，有时可能就是因为一点小bug过不去降低了整个程序的可用性，我什么时候才可以成为一个人肝一个程序的大佬啊！当然，团队还是最重要的，多一个人，多一份智慧，多一份力量（但大作业要独立完成）👊！

## 源代码🔊

### hush.h

```
/******  
hush.h  
本文件为哈希表（桶）hush的头文件  
构建了一个哈希表类（LZ77所用）和一个哈希桶类（huffman所用）  
哈希方式大大提高了查找的效率,为压缩文件节省了大量的时间  
*****/  
  
#define _CRT_SECURE_NO_WARNINGS  
  
#include<iostream>  
#include<string>  
  
using namespace std;  
  
typedef unsigned char USC;  
typedef unsigned short USH;  
typedef unsigned long long USL;  
  
const USH HWS = USH(32 * 1024); //hush table head and windows' size 哈希表head数组的大小和窗口大小的一半 32*1024=32K  
const USH MIN_MATCHLENGTH = 3; //最短的匹配长度为3  
const USH MAX_MATCHLENGTH = 258; //最长的匹配长度255+3=258  
const USH MIN_LOOKAHEAD = MAX_MATCHLENGTH + 1; //先行缓冲区的最小长度，当小于此长度时，就使窗口滑动32K  
const USH HASH_BITS = 15;  
  
const USH HUSHSIZE = USH(32 * 1024); //哈希表数组的大小  
const USH HUSHMASK = USH(HUSHSIZE - 1); //防止哈希表数组溢出  
  
//哈希表  
class HushTable {
```

```

public:
    HushTable(USH size);
    ~HushTable();
    void UpdateHush();
    void HushFunction(USH& hashAddr, USC ch);
    void HushInsert(USH& matchHead, USC ch, USH pos, USH& hashAddr);
    USH GetNextAdress(USH matchHead);
    USH HUSH_SHIFT();

private:
    USH* Head;          //哈希表的head数组，用以存放某（三字符）字符串的匹配头，索引为（三字符）字符串对应的哈希地址（通过哈希函数获取）
    USH* Prev;          /*哈希表的prev数组，用以存放上一个字符串的位置，用于解决哈希冲突（即字符串对应的哈希地址将原匹配头覆盖可能会使后续的匹配中得不到最优匹配）
                        将字符连成匹配链*/
};

```

## hush.cpp

```

#include "hush.h"

/*****
功能：  HushTable类的构造函数
返回值：  void
描述：  构造哈希表数组head和prev，申请含有2*HUSHSIZE个元素的空间
        将head和prev数组的内存连接起来，并且赋0
*****/
HushTable::HushTable(USH size) {

    Prev = new USH[size * 2];
    if (Prev == NULL) {
        cout << "Prev is no memory!\n";
    }
    Head = Prev + size;

    //将head和prev数组元素赋0
    memset(Prev, 0, size * 2 * sizeof(USH));

}

/*****
功能：  HushTable类的析构函数
返回值：  void
描述：  delete数组prev申请的含有2*HUSHSIZE个元素的空间
        将head和prev指针置空
*****/
HushTable::~~HushTable() {
    delete[] Prev;
    Prev = nullptr;
    Head = nullptr;
}

/*****
功能：  更新哈希表
返回值：  void
描述：  当窗口向前移动32K时（即先前缓冲区大小不足MIN_LOOKAHEAD）

```

需要将head数组中存放位置大于32K的元素减去32K，将存放的位置小于32K的元素置0

\*\*\*\*\*/

```
void HushTable::UpdateHush() {  
  
    for (USH i = 0; i < HUSHSIZE; i++) {  
  
        //更新head数组  
        if (Head[i] >= HWS) {  
            Head[i] -= HWS;  
        }  
        else {  
            Head[i] = 0;  
        }  
  
        //更新prev数组  
        if (Prev[i] >= HWS) {  
            Prev[i] -= HWS;  
        }  
        else {  
            Prev[i] = 0;  
        }  
    }  
}
```

\*\*\*\*\*/

功能： 获得字符串的哈希函数

参数： hashAddr 上一个串的哈希地址 ch 要计算哈希地址的字符串的第三个字符

返回值： void

描述： 通过传入字符串（三字符）的第三个字符来获取本字符串的哈希地址，（计算字符串哈希地址需要三个字符，只传入一个字符的原因是前两个字符的哈希地址已经被保存在上一个字符串的哈希地址中，所以只需要第三个字符就可以将该串的哈希地址计算出

\*\*\*\*\*/

```
void HushTable::HushFunction(USH& hashAddr, USC ch) {  
    hashAddr = (((hashAddr) << HUSH_SHIFT()) ^ (ch)) & HUSHMASK;  
}
```

\*\*\*\*\*/

功能： 通过字符串的哈希地址将其位置插入到哈希表中

参数： matchHead 匹配头 ch 要插入的字符串的第三个字符 pos 当前串的位置 hashAddr 上一个串的哈希地址

返回值： void

描述： 每从先行缓冲区读取一个字符时，就计算其与其后两个字符组成的字符串的哈希地址（head数组的索引），获得对应head数组元素（若为0说明无匹配，即之前没有出现该串），再将该元素赋给prev数组中索引为当前位置的元素，从而形成匹配链

\*\*\*\*\*/

```
void HushTable::HushInsert(USH& matchHead, USC ch, USH pos, USH& hashAddr) {  
    //通过哈希函数计算出当前串的哈希地址  
    HushFunction(hashAddr, ch);
```

//传入匹配链头（用于查找匹配串），再将head[hashAddr]的值（离本串最近的匹配串的位置）赋给匹配链头

```
    matchHead = Head[hashAddr];
```

```
    //将head[hashAddr]的值（离本串最近的匹配串的位置）赋给prev数组中索引为当前位置的元素，从而形成匹配链
```

```
    Prev[pos & HUSHMASK] = Head[hashAddr];
```

```
    //将当前位置赋给匹配链的头
```

```
    Head[hashAddr] = pos;
```

```
}
```

```
/******
```

```
功能： 遍历匹配链
```

```
参数： 当前匹配头
```

```
返回值： 链中下一个串的位置
```

```
描述： 传入匹配链的一个节点，通过prev的索引返回下一个匹配串的位置以再次进行匹配
```

```
*****/
```

```
USH HushTable::GetNextAddress(USH matchHead) {
```

```
    //通过prev的索引返回下一个匹配串的位置
```

```
    return Prev[matchHead & HUSHMASK];
```

```
}
```

```
USH HushTable::HUSH_SHIFT() {
```

```
    return (HASH_BITS + MIN_MATCHLENGTH - 1) / MIN_MATCHLENGTH;
```

```
}
```

## LZ77.h

```
/******
```

```
本头文件为LZ77（GZIP中的LZ77压缩方式）头文件，创建了一个LZ77类
```

```
附：（LZ77简要步骤，详细步骤会在实验报告中写明）
```

```
1.创建一个窗口（查找缓冲区和先行缓冲区）（64K=32K+32K）
```

```
2.随着查找缓冲区的前进和窗口的前进，不断构建字典
```

```
3.构建字典的同时，将先行缓冲区头三个字符组成的字符串与先前的字符串进行匹配（通过字典）  
为提高匹配效率，采用哈希表的方式进行匹配
```

```
4.若匹配成功，则通过哈希表构建的匹配链匹配出最长的匹配串，将当前串用<距离，长度>进行  
替换后输出到压缩文件；若匹配不成功则原样输出到压缩文件
```

```
5.最后将压缩文件和标记文件（标记字符是被替换的‘对’还是原字符）合并组成最终的压缩文件
```

```
*****/
```

```
#define _CRT_SECURE_NO_WARNINGS
```

```
#include <iostream>
```

```
#include "hush.h"
```

```
#include <fstream>
```

```
#include <string>
```

```
#include <cstdio>
```

```
using namespace std;
```

```
class LZ77 {
```

```
public:
```

```
    LZ77();
```

```
    ~LZ77();
```

```
    void CompressFile(string fileName, string newFileName);
```

```
    void UnCompressFile(string fileName, string newFileName);
```

```
    void Movewindow(ifstream& inFile, size_t& lookAhead, USH& strStart);
```

```

        USC GetMaxMatchLength(USH matchHead, USH& curMatchDistance, USH strStart);
        void WriteFlag(ofstream& flagFile, USC& chFlag, USC& bitCount, bool
hasEncode);
        void MergeFile(ofstream& fout, USL fileSize);

        void Writewindow(FILE* fOUT, size_t& pos);
private:
        USC* window;           //滑动窗口，大小为64k，用于存放缓冲区的数据
        HushTable LZhush;      //窗口对应的哈希表，随着窗口的滑动不断更新
};

```

## LZ77.cpp

```

#include "LZ77.h"

/*****
功能： LZ77类的构造函数
返回值： void
描述： 构造一个LZ77类，申请一个窗口大小（64k）的空间和一个哈希
      表
*****/
LZ77::LZ77()
    :window(new USC[2 * HWS]),
    LZhush(HWS) { }

/*****
功能： LZ77类的析构函数
返回值： void
描述： delete申请的一个窗口大小（64k）的空间
*****/
LZ77::~LZ77() {
    delete[] window;
    window = NULL;
}

/*****
功能： 核心压缩函数，将文件通过LZ77算法压缩
参数： fileName 文件名 压缩后的文件名
返回值： void
描述： 通过滑动窗口中的先行缓冲区读取的字符（和其后两个字符构成
      的字符串）建立哈希表，同时建立匹配链，再用当前串通过哈希
      表的匹配链查找最长匹配，将<长度，距离>对替换该串（当找不
      到匹配就不做替换，继续读入下一字符）；再将每个位置对应的
      标记信息（是否被‘<长度，距离>对’替换了）写入到标记文件
*****/
void LZ77::CompressFile(string fileName, string newFileName) {

    //打开要压缩的数据文件
    ifstream fIn(fileName, ios::in | ios::binary);
    if (!fIn.is_open()) {
        cout << "open file error!";
        exit(0);
    }

```

```

}

//打开压缩后的数据文件
ofstream fOut(newFileName, ios::out | ios::binary);
if (!fOut.is_open()) {
    cout << "open file error!";
    exit(0);
}

//打开临时的标记信息文件
ofstream fFlag("flag.txt", ios::out | ios::binary);
if (!fFlag.is_open()) {
    cout << "open file error!";
    exit(0);
}

//获取文件大小
fIn.seekg(0, ios::end);
USL fileSize = fIn.tellg();
fIn.clear();
fIn.seekg(0, ios::beg);

//先从文件中读取64K（32K+32K）个字节，并且获得当前读取的字节数作为先前缓冲区
fIn.read((char*)window, HWS * 2);
size_t lookAhead = size_t(fIn.gcount());

//首先利用前两个字符设置最初的哈希地址
USH hashAddress = 0;
LZhush.HushFunction(hashAddress, window[0]);
LZhush.HushFunction(hashAddress, window[1]);

//与查找匹配相关的变量
USH strStart = 0; //当前匹配查找的位置（即先前缓冲区的第一个字符在窗口中的
//相对位置）
USH matchHead = 0; //当前字符串匹配查找的头位置，若为0则没有找到匹配串（即离
//当前串距离最近的匹配串的位置）
USC curMatchLength = 0; //当前串的匹配长度
USH curMatchDistance = 0; //当前匹配的距离（当前串与匹配串之间的距离）

//与写入标记信息相关的变量
USC chFlag = 0; //8bit标记信息
USC bitCount = 0; //标记信息bit位

//开始查找匹配
while (lookAhead) {

    //插入三字符字符串中的最后一个字符以计算哈希地址，获取匹配头，完善匹配链
    LZhush.HushInsert(matchHead, window[strStart + 2], strStart,
hashAddress);

    //更新与当前匹配相关的变量
    curMatchLength = 0; //当前的匹配长度
    curMatchDistance = 0; //当前匹配的距离（当前串与匹配串之间的距离）

    //当获取的匹配头不为0，则通过GetLonggestMatchLength函数得到匹配链中的最长匹配长度
    if (matchHead) {

```

```

        curMatchLength = GetMaxMatchLength(matchHead, curMatchDistance,
strStart);
    }

    //验证是否真的在匹配链中找到了不小于MIN_MACHLENGTH（最短的匹配长度）的匹配串
    if (curMatchLength >= MIN_MATCHLENGTH) {

        //说明找到了有效匹配串，将<长度，距离>对写入压缩数据文件中
        USC length = curMatchLength - 3;           //<长度>
        USH distance = curMatchDistance;           //<距离>
        fOut.put(length);                           //写入长度
        fOut.write((char*)&distance, sizeof(distance)); //写入距离

        //写入标记文件
        writeFlag(fFlag, chFlag, bitCount, true);

        //替换后将被替换的串中所有三字符字符串插入到哈希表中
        for (USC i = 0; i < curMatchLength - 1; i++) {
            strStart++;           //后移一个字符
            LZhush.HushInsert(matchHead, window[strStart + 2], strStart,
hashAddress); //插入三字符字符串
        }
        strStart++;           //循环过程中少移了一次

        //更新先行缓冲区中剩余字符的个数
        lookAhead = lookAhead - curMatchLength;
    }
    else {

        //说明没有找到有效匹配串，将字符原样输出到压缩数据文件中
        fOut.put(window[strStart]); //写入字符
        strStart++;           //后移一个字符
        lookAhead--;           //先前缓冲区减1

        //写入标记文件
        writeFlag(fFlag, chFlag, bitCount, false);

    }

    //当先行缓冲区的长度小于最小长度时，窗口滑动
    if (lookAhead <= MIN_LOOKAHEAD) {
        Movewindow(fIn, lookAhead, strStart);
    }
}

//当最后的标记位数不够8bit
if (bitCount > 0 && bitCount < 8) {

    //直接写入标记文件
    chFlag <=<= (8 - bitCount);
    fFlag.put(chFlag);
}

//关闭标记文件
fFlag.close();

//合并压缩数据文件与标记文件形成最终压缩文件

```



```

MergeFile(fout, fileSize);

//关闭各文件
fin.close();
fout.close();

// 将用来保存标记信息的临时文件删除掉
remove("flag.txt");

}

/*****
功能： 解压函数，将文件解压
参数： fileName 文件名 newFileName 解压后的文件名
返回值： void
描述： 从标记文件中读取标记,如果当前标记是0,表示源字符,从压缩数
据文件中读取一个字节,直接写到解压缩之后的文件中;如果当前标
记是1,表示遇到<长度,距离>,从压缩数据文件中先读取一个字节
作为压缩的长度,再继续读取两个字节作为距离,然后从解压缩过
的结果中找出匹配长度
*****/
void LZ77::UnCompressFile(const string fileName,const string newFileName)
{
    // 操作压缩文件 "原始数据" 的指针
    FILE* fin = fopen(fileName.c_str(), "rb");
    if (nullptr == fin)
    {
        std::cout << "压缩文件打开失败" << std::endl;
        return;
    }

    // 操作压缩文件 "标记数据" 的指针
    FILE* finFlag = fopen(fileName.c_str(), "rb");
    if (nullptr == finFlag)
    {
        std::cout << "压缩文件打开失败" << std::endl;
        return;
    }

    // 获取原始文件的大小
    USL fileSize = 0;
    fseek(finFlag, 0 - int(sizeof(fileSize)), SEEK_END);
    fread(&fileSize, sizeof(fileSize), 1, finFlag);

    // 获取标记信息的大小
    USL flagSize = 0;
    fseek(finFlag, 0 - int(sizeof(fileSize)) - int(sizeof(size_t)), SEEK_END);
    fread(&flagSize, sizeof(fileSize), 1, finFlag);

    // 将标记信息文件的文件指针移动到保存标记数据的起始位置
    fseek(finFlag, 0 - int(sizeof(fileSize)) - int(sizeof(size_t)) -
long(flagSize), SEEK_END);

    //打开解压文件
    FILE* fout = fopen(newFileName.c_str(), "wb");

```

```

//用于从已经解压好的数据中向前读取重复的字节
FILE* fOutIn = fopen(newFileName.c_str(), "rb");

USH bitCount = 0;
USH chFlag = 0;
USH encodeCount = 0;    //已经解码的长度

while (encodeCount < fileSize){

    //处理完完8bit时，再读取8bit
    if (0 == bitCount)
    {
        chFlag = fgetc(fInFlag);
        bitCount = 8;
    }

    if (chFlag & 0x80)
    {
        // 距离长度对
        USH matchLength = fgetc(fIN) + 3;
        USH matchDistance = 0;
        fread(&matchDistance, sizeof(matchDistance), 1, fIN);

        // 清空缓冲区：系统会将缓冲区中的数据写入到文件中
        fflush(fOUT);

        // 更新已经解码字节数大小
        encodeCount += matchLength;

        //移动文件指针至匹配字符处
        fseek(fOutIn, 0 - matchDistance, SEEK_END);

        //写入匹配的字符
        for (size_t i = 0; i < matchLength; i++) {
            fputc((fflush(fOUT), fgetc(fOutIn)), fOUT);
        }
    }
    else
    {
        //写入原字符
        int ch = fgetc(fIN);
        fputc(ch, fOUT);
        encodeCount++;
    }

    //再读取一个bit位
    chFlag <<= 1;
    bitCount--;
}

//关闭相关文件
fclose(fIN);
fclose(fInFlag);
fclose(fOUT);
fclose(fOutIn);
}

```

```

/*****
功能： 获得最长的匹配串
参数： matchHead 匹配头 strStart 当前字符串的位置 curMatchDistance 当前串与匹配串的距离
返回值： 最长的匹配长度
描述： 通过匹配链不断查找得到最长匹配长度的匹配串，但是要通过限制匹配次数（次数可自定义，这影响了压缩的程度和时间）以防止陷入死循环
*****/
USC LZ77::GetMaxMatchLength(USH matchHead, USHORT curMatchDistance, USHORT strStart)
{
    //相关的变量
    USC nowMatchLength = 0; //每次匹配的长度
    USHORT curMatchStart = 0; // 当前匹配在查找缓冲区中的起始位置
    USC maxMatchTimes = 255; //最大的匹配次数
    USC maxMatchLength = 0; //最长的匹配长度

    do {
        //更新 从当前串开始的匹配范围（先行缓冲区中）
        USC* pStart = window + strStart; //开始
        USC* pEnd = pStart + MAX_MATCHLENGTH; //结束

        //更新 查找缓冲区的起始
        USC* pMatchStart = window + matchHead;

        //更新 每次匹配的长度
        nowMatchLength = 0;

        //逐字符进行匹配
        while (pStart < pEnd && *pStart == *pMatchStart) {

            //使两串再后移一位进行匹配，再使匹配长度加1
            pStart++;
            pMatchStart++;
            nowMatchLength++;
        }

        //一次匹配结束了，判断并记录最长匹配
        if (nowMatchLength > maxMatchLength) {
            maxMatchLength = nowMatchLength; //更新 长度
            curMatchStart = matchHead; //更新 距离
        }

        //当匹配头为空（即为0）时或者达到了最大的匹配次数时，停止匹配
    } while ((matchHead = LZHush.GetNextAddress(matchHead)) > 0 && maxMatchTimes--);

    //获得最终最长匹配对应的距离
    curMatchDistance = strStart - curMatchStart;

    //返回最长匹配长度
    return maxMatchLength;
}

/*****

```

功能： 滑动窗口

参数： **fIn** 要压缩文件 **lookAhead** 先前缓冲区剩余字节数 **strStart**  
当前串所在位置（即先前缓冲区的第一个字符位置）

返回值： **void**

描述： 将滑动窗口向后移动32K

\*\*\*\*\*/

```
void LZ77::Movewindow(ifstream& fIn, size_t& lookAhead, USH& strStart) {
```

```
    //先判断是否到达了文件末端
```

```
    if (strStart >= HWS) {
```

```
        //将右窗中的数据移至左窗
```

```
        memcpy(window, window + HWS, HWS);
```

```
        //将右窗置0
```

```
        memset(window + HWS, 0, HWS);
```

```
        strStart -= HWS;
```

```
        //更新哈希表
```

```
        Lzhush.updateHush();
```

```
        //向右窗（先前缓冲区）补充数据
```

```
        if (!fIn.eof()) {
```

```
            fIn.read((char*)window + HWS, HWS);
```

```
            lookAhead = lookAhead + size_t(fIn.gcount());
```

```
        }
```

```
    }
```

```
}
```

\*\*\*\*\*/

功能： 写入标记信息

参数： **fFlag** 标记文件 **chFlag** 8bit标记信息 **bitcount** bit位数  
**hasEncode** 判断是否已被替换

返回值： **void**

描述： 通过判断字符是否被<长度, 距离>对替换, 来向标记文件写入标记信息, 用0（一个bit）表示未被替换, 用1（一个bit）表示已被替换当满8bit时, 将**chFlag**写入标记文件中

\*\*\*\*\*/

```
void LZ77::writeFlag(ofstream& fFlag, USC& chFlag, USC& bitCount, bool hasEncode) {
```

```
    //左移一位
```

```
    chFlag <<= 1;
```

```
    //若被替换, 则记录1
```

```
    if (hasEncode) {
```

```
        chFlag |= 1;
```

```
    }
```

```
    bitCount++;
```

```
    //当标记满8bit时, 将chFlag写入标记文件
```

```
    if (bitCount == 8) {
```

```
        fFlag.put(chFlag);
```

```
        //更新chFlag
```

```
        chFlag = 0;
```

```

        //更新bit位数
        bitCount = 0;
    }
}

/*****
功能： 合并标记信息与压缩文件信息
参数： fOut 压缩数据文件 fileSize
返回值： void
描述： 将标记信息与压缩文件信息合并成最后的压缩文件
*****/
void LZ77::MergeFile(ofstream& fOut, USL fileSize) {

    //打开标记文件
    ifstream fFlag("flag.txt", ios::in | ios::binary);
    if (!fFlag.is_open()) {
        cout << "open file error!";
        exit(0);
    }

    size_t flagSize = 0; //标记文件的大小
    char* pReadBuff = new char[1024]; //读取缓冲区，每次读1024个字节
    if (pReadBuff == NULL) {
        cout << "pReadBuff is no memory!\n";
        return;
    }

    //读取标记文件内容
    while (true) {

        //读取缓冲区
        fFlag.read(pReadBuff, 1024);
        size_t readSize = size_t(fFlag.gcount());

        //若读取完毕，则退出循环
        if (readSize == 0) break;

        //将缓冲区的标记信息写入压缩文件
        fOut.write(pReadBuff, readSize);

        //记录标记文件大小
        flagSize = flagSize + readSize;
    }

    //写入标记文件大小
    fOut.write((char*)&flagSize, sizeof(flagSize));

    // 写原始文件大小
    fOut.write((char*)&fileSize, sizeof(fileSize));

    //关闭文件并delete缓冲区
    fFlag.close();
    delete[] pReadBuff;
}

```

# main.cpp

```
#include<iostream>
#include "LZ77.h"
#include<windows.h>

int main(int argc, char* argv[]) {

    //若参数不为4，则提示错误
    if (argc != 4) {
        cerr << "Please make sure the number of parameters is correct." << endl;
        return -1;
    }

    //如果第3个参数为zip则执行压缩程序
    if (!strcmp(argv[3], "zip")) {

        //计算压缩时间
        USL t = GetTickCount64();

        //执行压缩
        LZ77 lz;
        lz.CompressFile(argv[1], argv[2]);

        cout <<"Compress File Time: "<< (GetTickCount64() - t) / 1000.0 << "s";
        return 0;
    }
    //如果第3个参数为unzip则执行解压缩程序
    if(!strcmp(argv[3], "unzip")){

        //计算解压时间
        USL t = GetTickCount64();

        //执行解压
        LZ77 lz;
        lz.UnCompressFile(argv[1], argv[2]);

        cout << "Uncompress File Time: " << (GetTickCount64() - t) / 1000.0 <<
"s";
        return 0;
    }
    else {

        //命令错误
        cerr << "Unknown parameter!\nCommand list: zip unzip" << endl;
        return -1;
    }
}
```