**程序设计基础课程设计》实验报告**

班级：1803014

姓名：侯晨

学号：18150100002

所做题目：1\_1

**第1\_1题**

* 1. **需求分析**

尽管long long类型（或者在有些编译器中表示为\_\_int64.注意int64前面有2个下划线）能表示的数已经相当大了，但和一些天文数字比起来，它连屏幕的一行都占不到。很多实际数学问题其实是long long也无法解决的。

例如，围棋的走法有3的361次方，这个数字怎么显示出来？宇宙中原子的总和是10的80次方，怎么显示出来？这些都是语言本身的数据类型无法表示的。实际上，这些数字如果用二进制去储存的话，需要的空间也是特别大的。为了节约空间，同时让我们能够算出这些大数据，我们需要一种应用数组进行模拟计算的方法——高精度计算。

* 1. **算法详细设计**

**模块划分：**

加法函数 add()

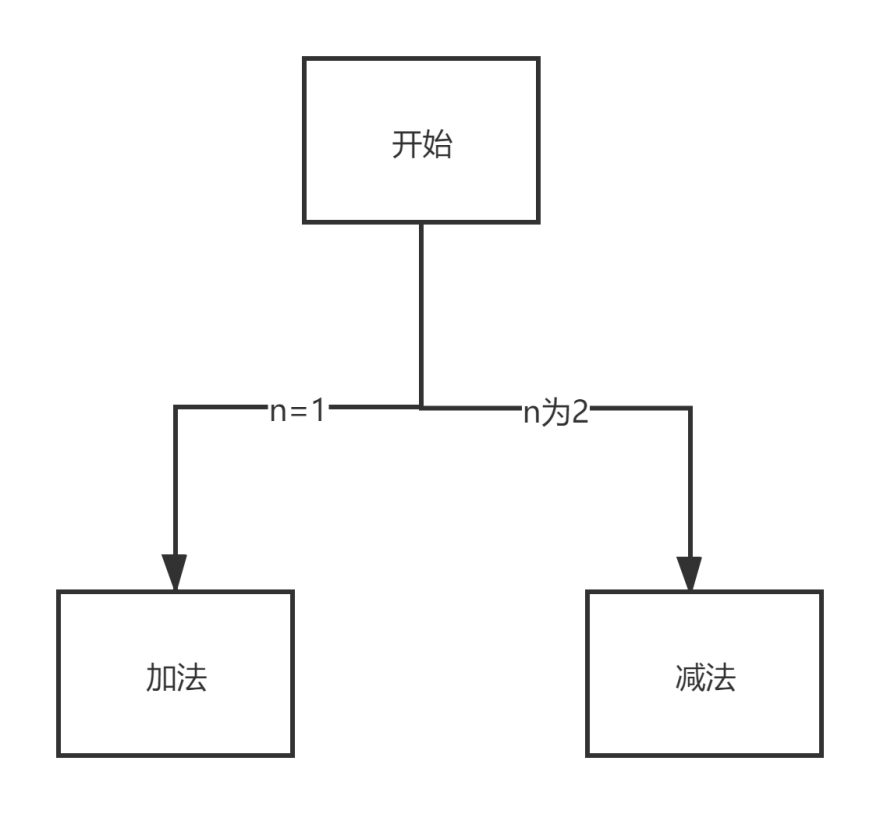
减法函数subtract()

快速幂函数 Pow(int a,int b)

**算法描述：**

通过先键入一个数字n,来判断是要进行加法还是减法，再键入两个数字进行运算，最后输出结果。

**流程图**

****

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

* 1. **算法详细设计源程序：**

no1\_1.c

#include<bits/stdc++.h>

using namespace std;

#define N 200

//整数乘幂运算函数

int Pow(int a, int b)

{

int i = 0, result = 1;

for(i = 0; i < b; ++i)

{

result \*= a;

}

return result;

}

int n;

void add(){

char stra[N], strb[N]; //字符串数组，以字符形式储存两个大数；

int i = 0, step = 4, carry = 0; //step表示块长，carry为进位位；

int lengtha, lengthb, maxlength, resultsize; //maxlength表示stra和strb二者长度较大的那个；

int numa[N], numb[N],numc[N]; //依次储存被加数，加数，和；

memset(numa, 0, sizeof(numa));

memset(numb, 0, sizeof(numb));

memset(numc, 0, sizeof(numc)); //初始化为零；

scanf("%s%s", stra, strb);

lengtha = strlen(stra);

lengthb = strlen(strb); //计算两个大数的长度

//字符数字转为四位一块的整数数字

for(i = lengtha-1; i >= 0; --i)

{

numa[(lengtha-1-i)/step] += (stra[i]-'0')\*Pow(10,(lengtha-1-i)%step);

}

for(i = lengthb-1; i >= 0; --i)

{

numb[(lengthb-1-i)/step] += (strb[i]-'0')\*Pow(10,(lengthb-1-i)%step);

}

maxlength = lengtha > lengthb ? lengtha : lengthb;

//逐块相加，并进位

for(i = 0; i <= maxlength/step; ++i)

{

numc[i] = (numa[i] + numb[i])%Pow(10, step) + carry; //计算和

carry = (numa[i] + numb[i])/Pow(10, step); //计算进位

}

//计算最后和的块的总数

resultsize = numc[maxlength/step] > 0 ? maxlength/step : maxlength/step - 1;

printf("%d", numc[resultsize]);

for(i = resultsize-1; i >= 0; --i)

{

printf("%04d", numc[i]); //右对齐，补零输出；

}

printf("\n");

}

void subtract(){

char stra[N], strb[N]; //字符串数组，以字符形式储存两个大数；

int i = 0, step = 4, borrow = 0, mark = 0; //step表示块长，borrow为借位位, mark为结果符号位；

int lengtha, lengthb, maxlength, resultsize; //maxlength表示stra和strb二者长度较大的那个；

int numa[N], numb[N],numc[N], \*maxnum, \*minnum; //依次储存被减数，减数，和；

memset(stra, 0, sizeof(stra));

memset(strb, 0, sizeof(strb));

memset(numa, 0, sizeof(numa));

memset(numb, 0, sizeof(numb));

memset(numc, 0, sizeof(numc)); //初始化为零；

scanf("%s%s", stra, strb);

lengtha = strlen(stra);

lengthb = strlen(strb); //计算两个大数的长度

maxlength = lengtha >= lengthb ? lengtha : lengthb;

//字符数字转为四位一块的整数数字

for(i = lengtha-1; i >= 0; --i)

{

numa[(lengtha-1-i)/step] += (stra[i]-'0')\*Pow(10,(lengtha-1-i)%step);

}

for(i = lengthb-1; i >= 0; --i)

{

numb[(lengthb-1-i)/step] += (strb[i]-'0')\*Pow(10,(lengthb-1-i)%step);

}

//找出较大的数

maxnum = numa;

minnum = numb;

mark = 1;

for(i = (maxlength-1)/step; i >= 0; --i)

{

if(numa[i] > numb[i])

{

maxnum = numa;

minnum = numb;

mark = 1;

break;

}

else if(numa[i] < numb[i])

{

maxnum = numb;

minnum = numa;

mark = -1;

break;

}

}

//逐块相减，并借位

for(i = 0; i <= maxlength/step; ++i)

{

numc[i] = (maxnum[i] - minnum[i] + Pow(10, step) + borrow)%Pow(10,step); //计算差

borrow = (maxnum[i] - minnum[i] + Pow(10, step) + borrow)/Pow(10, step) - 1; //计算借位

}

//计算最后和的块的总数

resultsize = maxlength/step;

while(!numc[resultsize]) --resultsize;

printf("%d", mark\*numc[resultsize]);

for(i = resultsize-1; i >= 0; --i)

{

printf("%04d", numc[i]); //右对齐，补零输出；

}

printf("\n");

}

int main(){

printf("加法：1，减法：2\n");

cin>>n;

if(n==1) add();

if(n==2) subtract();

return 0;

}

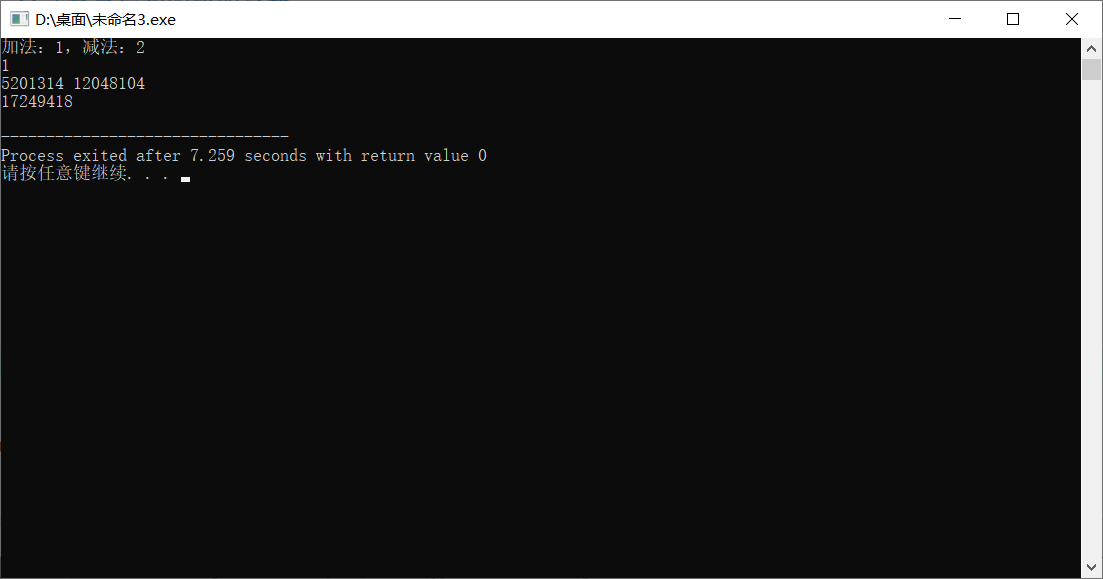
* 1. **主要函数设计描述**

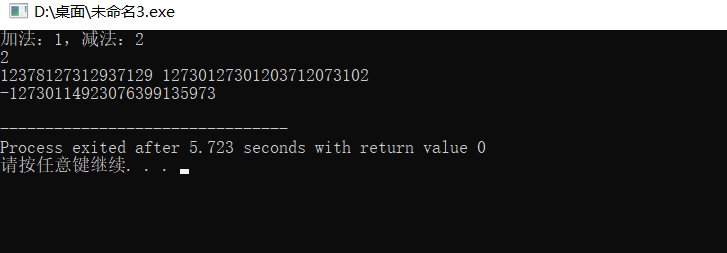
加法函数 add()

减法函数subtract()

快速幂函数 Pow(int a,int b)

* 1. **测试数据（输入、输出）:**





第1\_2题

* 1. 需求分析

编写一个程序模拟堆栈，要求能够模拟、入栈、出栈、返回栈顶元素等基本操作。栈中元素可用整数代替。不能使用C++模板库预定义的类型。程序运行中可输入多组入栈、出栈操作，每次操作后展示栈中元素。

* 1. 算法详细设计

**算法描述**

实现一个基本的栈，可以使用双端链表的方式，这样就可以记录下前一个节点的数据，也就是可以确保能够会到栈顶的上一层。

**模块划分**

struct Node

struct stack

stack\* inintStack()

int isEmpty()

int getSize()

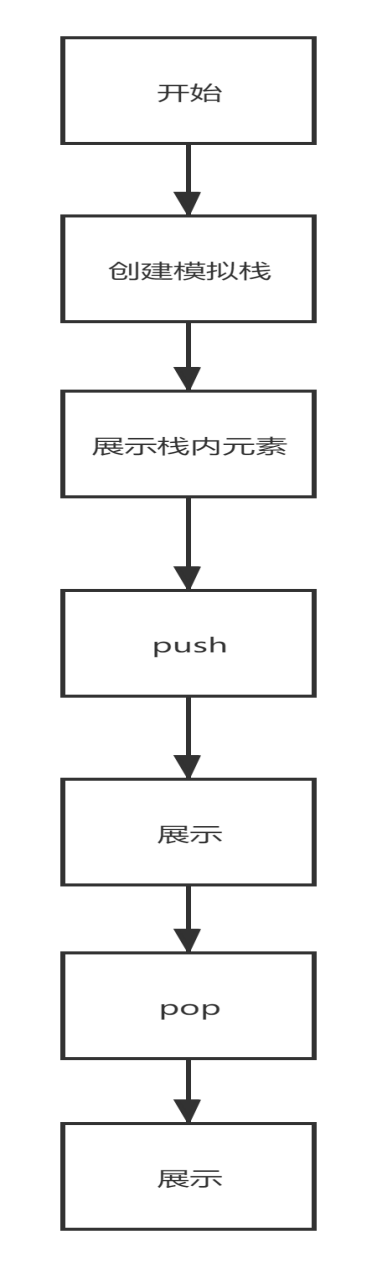
int getTop()

void push()

void show()

void pop()

**流程图**



* 1. **算法详细设计源程序：**

**no1\_2.c**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int val = 0;

Node\* pre = NULL;

Node\* next = NULL;

};

struct stack {

Node\* top = NULL;//记录下当前栈顶

int size = 0;

Node\* node = NULL;

};

//在栈中嵌套链表，并使用size 变量记录下栈的大小

stack\* inintStack() {

stack\* sta = (stack\*)malloc(sizeof(stack));

sta->node = (Node\*)malloc(sizeof(Node));

sta->size = 0;

sta->top = NULL;

return sta;

}

int isEmpty(stack\* sta) {

return sta->size == 0;

}

int getSize(stack\* sta) {

return sta->size;

}

int getTop(stack\* sta) {

return sta->size == 0 ? -1 : sta->top->val;

}

void push(stack\* sta, int val) {

sta->size++;

//创建新节点

sta->node->next = (Node\*)malloc(sizeof(Node));

sta->node->next->val = val;

//连接

sta->node->next->pre = sta->node;

sta->node = sta->node->next;

//下一节点置空，更新top

sta->node->next = NULL;

sta->top = sta->node;

}

void show(stack\* sta) {

Node\* cur = sta->node;

int size = sta->size;

printf("stack: ");

if (size == 0)

printf("NULL");

while (size-- > 0) {

printf("%d\t", cur->val);

cur = cur->pre;

}

printf("\n");

}

int pop(stack\* sta) {

if (sta->size == 0)

return -1;

sta->size--;

sta->node = sta->node->pre;

free(sta->node->next);

sta->top = sta->node;

return 1;

}

int main() {

//测试的一些操作

stack\* sta = inintStack();

show(sta);

push(sta, 1); push(sta, 2); push(sta, 3);show(sta);

pop(sta); show(sta);

pop(sta); show(sta);

push(sta, 4); show(sta);

pop(sta); show(sta);

pop(sta); show(sta);

return 0;

}

* 1. **主要函数设计描述**

stack\* inintStack() 初始化栈

int isEmpty(stack) 判断栈是否为空 1为非空 0为空

int getSize()获取栈的大小 返回值为栈的大小

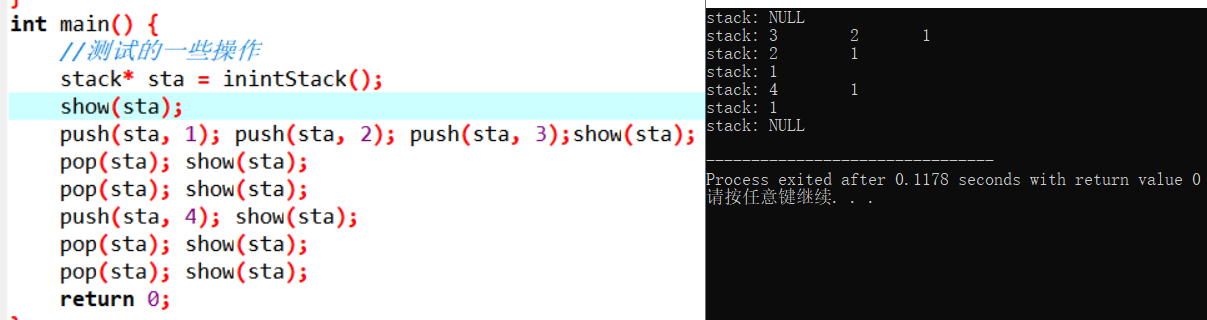
int getTop() 获取栈顶元素 返回值为栈顶得元素

void push(int) 将一个元素压入栈顶

void show() 展示栈中所有元素

void pop() 将栈顶元素弹出

* 1. **测试数据（输入、输出）：**



1\_3

* 1. **需求分析**

编写一个程序，可以在命令行输入参数，完成指定文件的缩放，并存储到新文件，命令行参数如下

zoom file1.bmp 200 file2.bmp

第一个参数为可执行程序名称，第二个参数为原始图像文件名，第三个参数为缩放比例（百分比），第四个参数为新文件名

* 1. **算法详细设计**

**算法分析：**

转化输入的压缩参数便于后续处理后将命令行参数传入zoom函数进行处理

传入原图图像名。先跳过位图文件头,然后定义位图信息头结构体变量head。从文件读入位图信息头结构体到head并从中获取图像的高,宽和每像素所占的位数。然后计算原图像每行字节数(注意位图数据中每个扫描行的字节数必须是4的倍数,这里+3是为了处理不满足4的倍数的情况,如果是4的倍数则结果和不+3的结果一样,如果不是4的倍数则结果进1位,先乘4然后除以4则是为了把数据归为4的倍数)。然后定义一个矩阵用以存放原图像像素信息,指针pIBuf指向该矩阵首地址。然后将源文件像素信息读入plBuf,单个元素大小为1字节,元素个数为每行字节数\*行数。

根据命令行参数设定新图像的长度和宽度(加0.5是由于强制转换时不四舍五入而是直接截去小数部分),新图像的每行字节数。然后定义一个矩阵用以存放新图像像素信息,指向该矩阵首地址。这些准备工作完成后双重循环遍历新图像的每一个像素点进行双向线性插值。完成像素数据填充后传入新图像的名字,存储新图像像素数据的矩阵和新图像的高度宽度,每像素位数进行保存操作。保存成功后释放像素矩阵指针。

先计算新图像每行字节数。打开文件写入图像数据。

**模块划分**

1、定义一个存储头文件数据结构体

typedef struct tagBITMAPFILEHEADER

{

 unsigned short bfType;      //保存图片类型。 'BM'

 unsigned long  bfSize;      //位图文件的大小，以字节为单位（3-6字节，低位在前）

 unsigned short bfReserved1;//位图文件保留字，必须为0(7-8字节）

 unsigned short bfReserved2;//位图文件保留字，必须为0(9-10字节）

 unsigned long  bfOffBits;  //RGB数据偏移地址,位图数据的起始位置，以相对于位图（11-14字节，低位在前）

}BITMAPFILEHEADER;

2定义一个存储位图信息的结构体

typedef struct tagBITMAPINFOHEADER

{

 unsigned long  biSize;      //本结构所占用字节数（15-18字节）

 unsigned long  biWidth;     //位图的宽度，以像素为单位（19-22字节）

 unsigned long  biHeight;    //位图的高度，以像素为单位（23-26字节）

 unsigned short biPlanes;    //目标设备的级别，必须为1(27-28字节）

 unsigned short biBitCount;  //每个像素所需的位数，必须是1（双色）（29-30字节）,4(16色），8(256色）16(高彩色)或24（真彩色）之一

 unsigned long  biCompression;//位图压缩类型，必须是0（不压缩），（31-34字节）

 //1(BI\_RLE8压缩类型）或2(BI\_RLE4压缩类型）之一

 unsigned long  biSizeImage;  //位图的大小(其中包含了为了补齐行数是4的倍数而添加的空字节)，以字节为单位（35-38字节）

 unsigned long  biXPelsPerMeter;//位图水平分辨率，每米像素数（39-42字节）

 unsigned long  biYPelsPerMeter;//位图垂直分辨率，每米像素数（43-46字节)

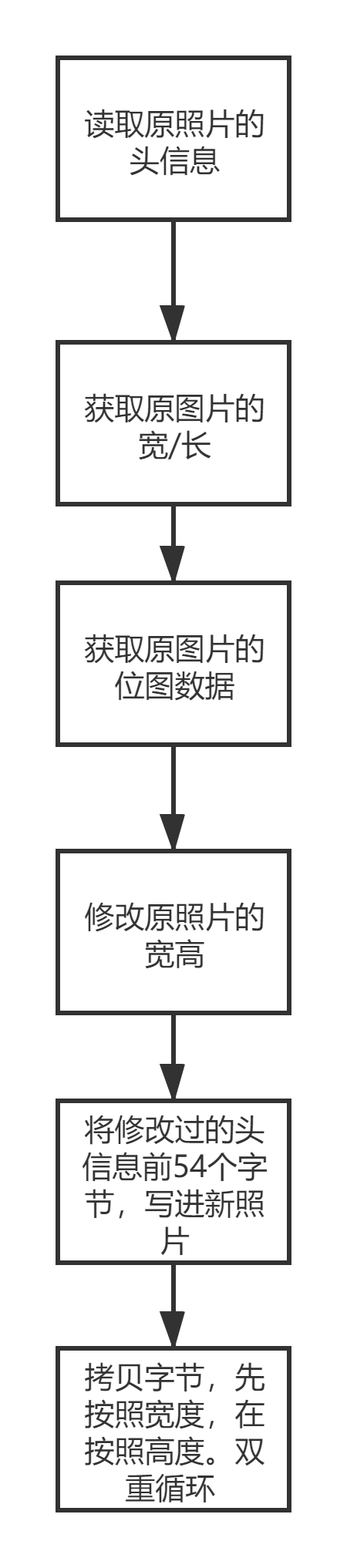
 unsigned long  biClrUsed;      //位图实际使用的颜色表中的颜色数（47-50字节）

 unsigned long  biClrImportant; //位图显示过程中重要的颜色数（51-54字节）

}BITMAPINFOHEADER;

3、图片缩小的原则就是按照一定的比例从范围内的像素点中抽去像素点。而放大的原则正好相反，将一个或多个像素点按照比例复制在其周围。

**流程**

****

* 1. **源代码**

**no1\_3.c**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include <stdlib.h>

#pragma pack(1) /\* 必须在结构体定义之前使用,这是为了让结构体中各成员按1字节对齐\*/

typedef struct tagBITMAPFILEHEADER

{

unsigned short bfType; //保存图片类型。 'BM'(1-2字节)

unsigned long bfSize; //位图文件的大小，以字节为单位（3-6字节，低位在前）

unsigned short bfReserved1;//位图文件保留字，必须为0(7-8字节）

unsigned short bfReserved2;//位图文件保留字，必须为0(9-10字节）

unsigned long bfOffBits; //RGB数据偏移地址,位图数据的起始位置，以相对于位图（11-14字节，低位在前）

}BITMAPFILEHEADER;

typedef struct tagBITMAPINFOHEADER

{

unsigned long biSize; //本结构所占用字节数（15-18字节）

unsigned long biWidth; //位图的宽度，以像素为单位（19-22字节）

unsigned long biHeight; //位图的高度，以像素为单位（23-26字节）

unsigned short biPlanes; //目标设备的级别，必须为1(27-28字节）

unsigned short biBitCount; //每个像素所需的位数，必须是1（双色）（29-30字节）,4(16色），8(256色）16(高彩色)或24（真彩色）之一

unsigned long biCompression;//位图压缩类型，必须是0（不压缩），（31-34字节）

//1(BI\_RLE8压缩类型）或2(BI\_RLE4压缩类型）之一

unsigned long biSizeImage; //位图的大小(其中包含了为了补齐行数是4的倍数而添加的空字节)，以字节为单位（35-38字节）

unsigned long biXPelsPerMeter;//位图水平分辨率，每米像素数（39-42字节）

unsigned long biYPelsPerMeter;//位图垂直分辨率，每米像素数（43-46字节)

unsigned long biClrUsed; //位图实际使用的颜色表中的颜色数（47-50字节）

unsigned long biClrImportant; //位图显示过程中重要的颜色数（51-54字节）

}BITMAPINFOHEADER;

void Bmp\_Bigger\_And\_Smaller(BITMAPFILEHEADER head,BITMAPINFOHEADER info,double bili,char \*a,char \*b)

{

FILE \*fpr1=fopen(a,"rb");

FILE \*fpw2=fopen(b,"wb");

if(fpr1==NULL||fpw2==NULL)

{

printf("图片打开失败!\n");

return ;

}

//读取原照片的头信息

fread(&head,sizeof(BITMAPFILEHEADER),1,fpr1);

fread(&info,sizeof(BITMAPINFOHEADER),1,fpr1);

unsigned int old\_width=info.biWidth;//获取原图片的宽

unsigned int old\_height=info.biHeight;//获取原图片的高

//获取原图片的位图数据

unsigned char \*src\_data=(unsigned char \*)malloc(old\_width\*old\_height\*3);

fseek(fpr1,54,SEEK\_SET); //随机定位，以字节数为1开始定位，后54位

fread(src\_data,old\_width\*old\_height\*3,1,fpr1);

printf("原图片的宽:%d\n",old\_width);

printf("原图片的高:%d\n",old\_height);

//修改原照片的宽高

unsigned int new\_width,new\_height;

printf("新图片的宽:%d \n",(int)bili\*old\_width);

printf("新图片的高:%d \n",(int)bili\*old\_height);

new\_width=(int)bili\*old\_width;

new\_height=(int)bili\*old\_height;

head.bfSize=new\_width\*new\_height\*3+54;

info.biWidth=new\_width;

info.biHeight=new\_height;

//将修改过的头信息前54个字节，写进新照片

fwrite(&head,sizeof(BITMAPFILEHEADER),1,fpw2);

fwrite(&info,sizeof(BITMAPINFOHEADER),1,fpw2);

//现在是把内容字节缩放，并且拷贝到put\_data

int i=0,j=0;

unsigned long dwsrcX,dwsrcY;

unsigned char \*pucDest;

unsigned char \*pucSrc;

unsigned char \*dest\_data=(unsigned char \*)malloc(new\_width\*new\_height\*3);

//这个拷贝字节数的作用，先按照宽度，在按照高度。双重循环

for(i=0;i<new\_height;i++)

{

dwsrcY=i/bili;

pucDest=dest\_data+i\*new\_width\*3; //下面完整的for循环，所存入的字节量

pucSrc=src\_data+dwsrcY\*old\_width\*3; //缩小/放大了

for(j=0;j<new\_width;j++)

{

dwsrcX=j/bili;

memcpy(pucDest+j\*3,pucSrc+dwsrcX\*3,3);//数据拷贝

//从源sourec中复制m个字节到目标destination中 ,(void\* destination, void\* sourec, unsigned m);

}

}

fseek(fpw2,54,SEEK\_SET); //随机定位，以字节数为1开始定位，后54位

fwrite(dest\_data,new\_width\*new\_height\*3,1,fpw2);

printf("成功!\n");

//释放堆空间,关闭文件

free(dest\_data);

free(src\_data);

fclose(fpr1);

fclose(fpw2);

}

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

{

//定义原照片信息结构体

BITMAPFILEHEADER old\_head;

BITMAPINFOHEADER old\_info;

//将结构体清空

memset(&old\_head,0,sizeof(BITMAPFILEHEADER));

memset(&old\_info,0,sizeof(BITMAPINFOHEADER));

double bili = atof(argv[2])/100.0;//把字符串转化为double型

printf("%f",bili);

Bmp\_Bigger\_And\_Smaller(old\_head,old\_info, bili,argv[1],argv[3]);

return 0;

}

* 1. **主要函数设计描述**

void Bmp\_Bigger\_And\_Smaller(BITMAPFILEHEADER head,BITMAPINFOHEADER info,double bili,char \*a,char \*b)

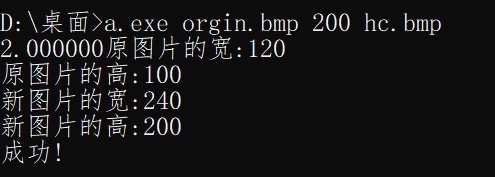
进行bmp大小转换

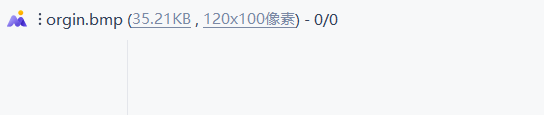
BITMAPFILEHEADER head,BITMAPINFOHEADER info 定义原照片信息结构体

Bili 转换比例

char \*a,char \*b 源文件与目的文件路径

* 1. 测试数据







1\_4

**1.1 需求分析：**

这道题要求是用RLE压缩解压算法，所以先要对其原理有一个大概的了解。

RLE全称（run-length encoding），翻译为游程编码，又译行程长度编码，又称变动长度编码法（run coding），在控制论中对于二值图像而言是一种编码方法，对连续的黑、白像素数(游程)以不同的码字进行编码。游程编码是一种简单的非破坏性资料压缩法，其好处是加压缩和解压缩都非常快。其方法是计算连续出现的资料长度再进行压缩。其优点是无损压缩，既节省了磁盘空间又不损失任何图像数据。但是当内容像ABCABCABC的话使用这种算法文件会增大，就是1A1B1C1A1B1C1A1B1C了，会变得更长，就达不到压缩的效果了。

**1.2 算法设计**

**算法描述：**

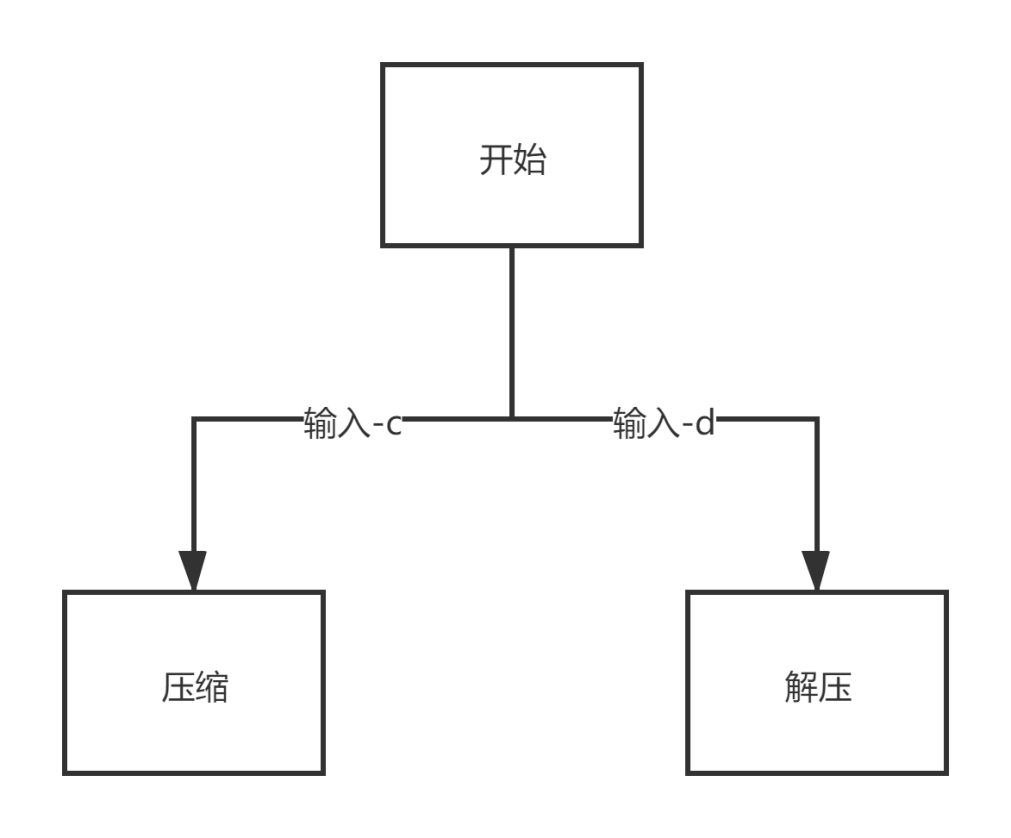
主要还是RLE算法本身吧， RLE是一种简单的压缩算法，主要用于压缩图像中连续的重复的颜色块。当然RLE并不是只能应用于图像压缩上，RLE能压缩任何二进制数据。原始图像文件的数据有一个特点，那就是有大量连续重复的颜色数据，RLE正好是用来压缩有大量连续重复数据的压缩编码，但对于其他二进制文件而言，由于文件中相同的数据出现概率较少，使用RLE压缩这些数据重复性不强的文件效果不太理想，有时候压缩后的数据反而变大了。RLE压缩方案是一种极其成熟的压缩方案，其特点是无损失压缩。

**模块划分：**

void unzip(char \*filename,char \*outfile) {//解压函数

void zip(char \*filename,char \*outfile) {//压缩函数

**流程图：**

****

**1.3 源代码**

**No.1\_4.c**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

void zip(char \*filename,char \*outfile)

{//压缩函数

FILE \*in, \*out;//定义指向文件的指针

int filelen;//每一个字符连续重复出现的次数

char cur,tmp;

if(!(in=fopen(filename,"rb")))//以二进制方式打开只读文件（原文件必须存在）

printf("文件打开失败\n");//若文件不存在则进行提示

else

{

out=fopen(outfile,"wb");/\*二进制方式打开只写文件

（若文件不存在则会进行创建，若存在则会覆盖原内容）\*/

cur=fgetc(in);//读取第一个字符，返回读取到的字符

tmp=cur;

filelen=1;

while(!feof(in))//读到文件末尾时结束循环

{

cur=fgetc(in);

if(cur==tmp){

filelen++;

}//进行RLE压缩

else{

fputc(filelen+'0',out);//写一个字符（此处是计数数字），若失败则返回EOF

fputc(tmp,out);//同上，此处是原字符

tmp=cur;

filelen=1;

}

}

}

fclose(in);

fclose(out);//关闭文件

}

void unzip(char \*filename,char \*outfile)

{//解压函数 ，注释会基本和上面一样 ，不再赘述

FILE \*in, \*out;

int filelen;

char cur;

if(!(in=fopen(filename,"rb")))

printf("文件打开失败\n");

else

{

out = fopen(outfile,"wb");

while(!feof(in)){

filelen = fgetc(in)-'0';

if(feof(in)) break;

cur = fgetc(in);

while(filelen--)

fputc(cur,out);

}

}

fclose(in);

fclose(out);

}

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

{//通过判断命令行参数进行相应操作

if(!strcmp(argv[2], "-d"))

{

unzip(argv[1], argv[3]);

printf("decompress finished\n");

}

else if(!strcmp(argv[2],"-c"))

{

zip(argv[1], argv[3]);

printf("compress finished\n");

}

else

printf("输入参数有误,请重新检查,-c : compress; -d : decompress\n");

return 0;

}

**1.4 主要函数设计描述**

void unzip(char \*filename,char \*outfile) {//解压函数

void zip(char \*filename,char \*outfile) {//压缩函数

**测试数据（输入、输出）:**

