

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**实验名称： 数据的表示**

**院 系 ：计算机科学与技术**

**专业班级 ： 计算机202203**

**学 号 ： U202215643**

**姓 名 ： 王国豪**

**指导教师 ： 朱 虹**

**2024 年 3 月 15 日**

**一、实验目的与要求**

⑴ 熟练掌握程序开发平台(VS2019/GCC+GDB) 的基本用法，包括程序的编译、链接和调试；

⑵ 熟悉地址的计算方法、地址的内存转换；

⑶ 熟悉数据的表示形式；

**二、实验内容**

**任务1 数据存放的压缩与解压编程**

定义了 结构 student ，以及结构数组变量old\_s[N], new\_s[N]; (N=5)

struct student {

char name[8];

short age;

float score;

char remark[200]; // 备注信息

};

编写程序，输入N个学生的信息到结构数组old\_s中。将 old\_s[N] 中的所有信息依次紧凑(压缩)存放到一个字符数组message中，然后从 message 解压缩到结构数组 new\_s[N]中。打印压缩前(old\_s)、解压后(new\_s)的结果，以及压缩前、压缩后存放数据的长度。

要求：

1. 输入的第0个人姓名(name)为自己的名字，分数为学号的最后两位；
2. 编写指定接口的函数完成数据压缩

压缩函数有两个： int pack\_student\_bytebybyte(student\* s, int sno, char \*buf);

int pack\_student\_whole(student\* s, int sno, char \*buf);

。pack\_student\_bytebybyte要求一个字节一个字节的向buf中写数据；pack\_student\_whole要求对short、float字段都只能用一条语句整体写入，用strcpy实现串的写入。

1. 使用指定方式调用压缩函数

old\_s数组的前N1（N1=2）个记录压缩调用pack\_student\_bytebybyte 完成；后N2（N2==3）个记录压缩调用pack\_student\_whole，两种压缩函数都只调用1次。

（4） 使用指定的函数完成数据的解压

解压函数的格式：int restore\_student(char \*buf, int len, student\* s);

buf 为压缩区域存储区的首地址；len为buf中存放数据的长度；s为存放解压数据的结构数组的起始地址； 返回解压的人数。解压时不允许使用函数接口之外的信息（即不允许定义其他全局变量）

（5）仿照调试时看到的内存数据，以十六进制的形式，输出message的前20个字节的内容，并与调试时在内存窗口观察到的message的前20个字节比较是否一致。

（6）对于第0个学生的score，根据浮点数的编码规则指出其各部分的编码，并与观察到的内存表示比较，验证是否一致。  
 (7) 指出结构数组中个元素的存放规律，指出字符串数组、short类型的数、float型的数的存放规律。

**任务2 编写数据表示的自动评测程序**

按照要求完成给定的功能，并**自动判断程序**的运行结果是否正确。（从逻辑电路与门、或门、非门等等角度，实现CPU的常见功能。所谓自动判断，即用简单的方式实现指定功能，并判断两个函数的输出是否相同。）

1. int absVal(int x); 返回 x 的绝对值

仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 10次

判断函数： int absVal\_standard(int x) { return (x < 0) ? -x : x;}

1. int negate(int x); 不使用负号，实现 -x

判断函数： int netgate\_standard(int x) { return -x;}

1. int bitAnd(int x, int y); 仅使用 ~ 和 |，实现 &

判断函数： int bitAnd\_standard(int x, int y) { return x & y;}

1. int bitOr(int x, int y); 仅使用 ~ 和 &，实现 |
2. int bitXor(int x, int y); 仅使用 ~ 和 &，实现 ^
3. int isTmax(int x); 判断x是否为最大的正整数（7FFFFFFF），

只能使用 !、 ~、 &、 ^、 |、 +

1. int bitCount(int x); 统计x的二进制表示中 1 的个数

只能使用，! ~ & ^ | + << >> ，运算次数不超过 40次

1. int bitMask(int highbit, int lowbit); 产生从lowbit 到 highbit 全为1，其他位为0的数。例如bitMask(5,3) = 0x38 ；要求只使用 ! ~ & ^ | + << >> ；运算次数不超过 16次。
2. int addOK(int x, int y); 当x+y 会产生溢出时返回1，否则返回 0

仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 20次

1. int byteSwap(int x, int n, int m); 将x的第n个字节与第m个字节交换，返回交换后的结果。 n、m的取值在 0~3之间。  
   例：byteSwap(0x12345678, 1, 3) = 0x56341278

byteSwap(0xDEADBEEF, 0, 2) = 0xDEEFBEAD

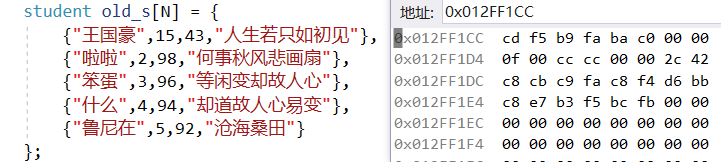
仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 25次

**三、实验记录及问题回答**

**（1）任务 1 的算法思想、运行结果等记录**

**算法思想：**

任务 1 的难点就是如何去理解压缩和解压的具体含义，简单的来说就是懂得不同数据类型在计算机中的存储方式，然后将数据压缩存储到一个char类型的数组message中，这个过程中可以把之前没有利用完的存储空间如name数组和remark数组未被利用的字节以及short类型和float类型之间为了对齐多的2个字节去掉，最后解压的时候再从message数组中将数据读取到新的结构体数组new\_s中，这就是基本的一个流程，下面来直观的看一下结构体数组中的存储。



我们很直观的看出了有很多内存没有被使用，我们就需要去压缩这部分的空间了。我们来具体实现一下压缩函数和解压函数

**pack\_student\_bytebybyte函数实现：**

要求我们要一个字节一个字节去压缩，我们可以通过(char\*)类型的指针遍历结构成员的内存，将具有实际意义的信息复制到新的内存段上，有一些细节需要注意，遍历字符串的时候遇到结束符‘\0’的时候停止遍历，并将指针加上对应的偏移量，然后读取short和float的时候注意用于内存对齐的两个字节。

****

**pack\_student\_whole函数实现：**

pack\_student\_whole要求对short、float字段都只能用一条语句整体写入，用strcpy实现串的写入，这样我们可以使用指针的强制类型转换来获取old\_s中的short和float，首先定义一个student类型的指针接收我们需要压缩的结构体数组，这样方便我们使用ppp[cnts].age来定位到我们需要压缩数据的位置，看一个例子：\*((short\*)pp) = ppp[cnts].age，我们还可以使用strcpy快速接收字符strcpy(pp,ppp[cnts].name)，值得注意的是strcpy函数是遇到结束符号’\0’停止，我们可以巧妙利用这个来压缩。

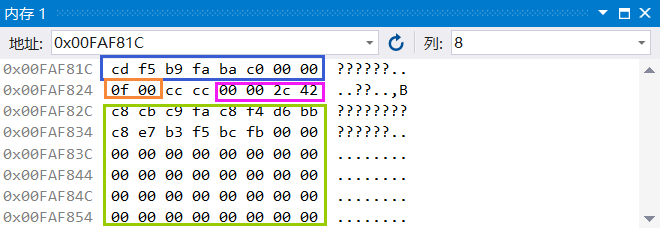
****

**restore\_student函数实现：**

这个函数我们直接利用strcpy拷贝字符串，利用指针类型强制转换来拷贝short和float变量，但是要注意的是解压的时候别忘记了内存对齐的两个字节。

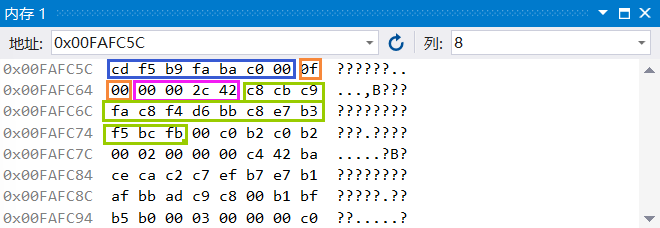
**压缩解压全过程验证分析：**

通过调试我们查询得到old\_s和message以及new\_s的地址，下面展示压缩前结构体数组中的内存信息：



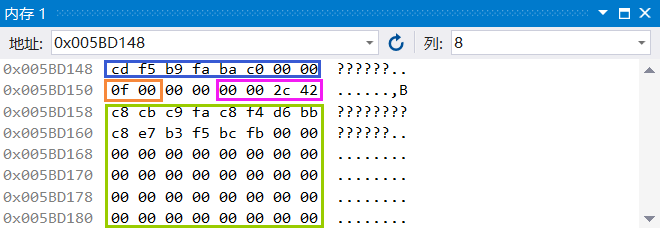
蓝色方框中的是name中的信息，橙色方框中的是age，紫色中的是score，绿色中的是remark的部分信息，我们可以很清楚的看到很多内存都没有被充分利用。（注：在我的vs2019中内存对齐的两个字节中存储的信息是 cc）

接下来我们看一下压缩后message中的内存信息：



可以把message中存储的内容和上图中的信息进行比较，发现内容一致，但是少了冗余的空间，说明我们的压缩是顺利的。

下面我们来看一下解压后的信息存储:



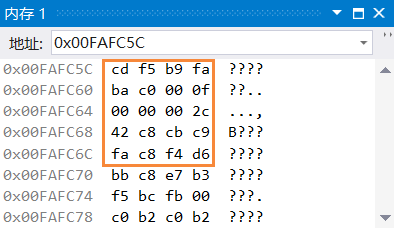
对照之前的信息，完美重合，证明解压过程没有问题。

**message的前20个字节的内容分析与验证：**

以十六进制的形式，输出message的前20个字节的内容的截图如下：



下面我们在vs的调试窗口查看message的前20个内容：



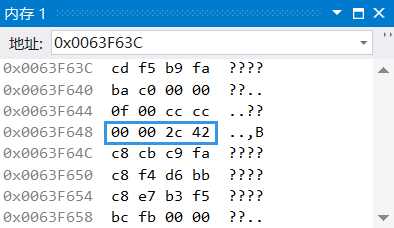
对照可知，完全匹配！验证成功。

**对于第0个学生的score验证浮点数编码：**

在我的数据中，第0个学生的score是43，按照单精度浮点数编码我们写出编码过程：

43的二进制编码为101011 = ，所以符号位s = 0, 阶码 = 5 + 127 = 1000 0100 故最后的32位编码为 0100 0010 0010 1100 0000 0000 0000 0000，转换成十六进制为 0x422c0000

我们再去观察调试中的内存信息：



有一点值得注意的是，高字节放在高地址，低字节放在低地址，通过匹配我们可以得知验证正确！

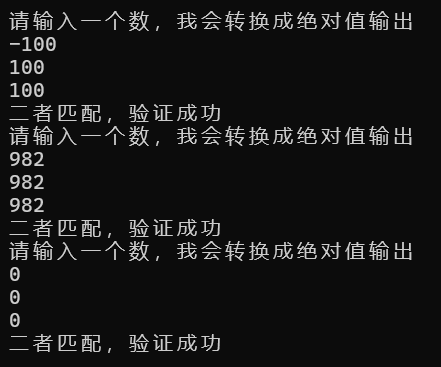
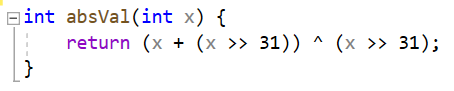
**结构数组中个元素的存放规律探究：**

通过以上实验，我们可以确定，结构体每个成员相对于结构体首地址的偏移量都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节。就例如我们的结构体中的short和float之间为了内存对齐，填充了两个字节。

**（2）任务 2 的算法思想、运行结果等记录**

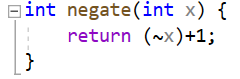
**int absVal(int x)函数实现分析：**

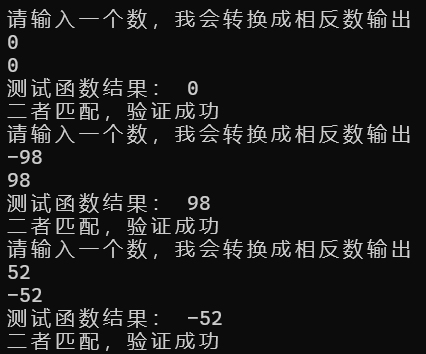
求绝对值我们需要判断形参x是正数还是负数，这二者的区别就是符号位，我们可以用x>>31来获取最高位的信息，注意的是这里是算术右移，如果x是负数，那么x>>31得到的是0xffffffff，我们就可以先减一后取反就实现了变成正数，如果x是正数的话，那么移位后得到的是0，后续的运算没有影响。

****

**int negate(int x)函数实现分析：**

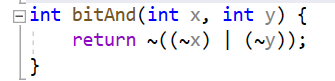
很简单的一个思路，直接取反加一。

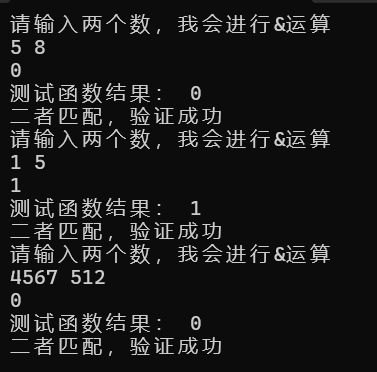
****

****

**int bitAnd(int x, int y)函数实现分析：**

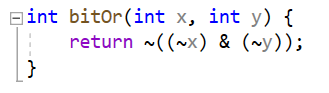
可以把|理解为数字电路中的或门，& 理解为与门，就可以很轻松得出答案。

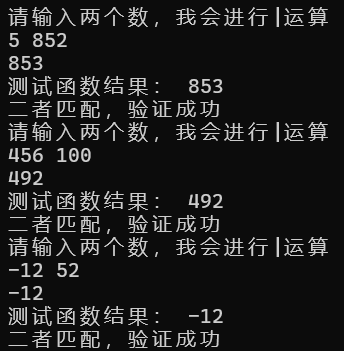




**int bitOr(int x, int y)函数实现分析：**

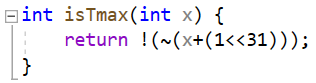
和上面的实现思路一样。

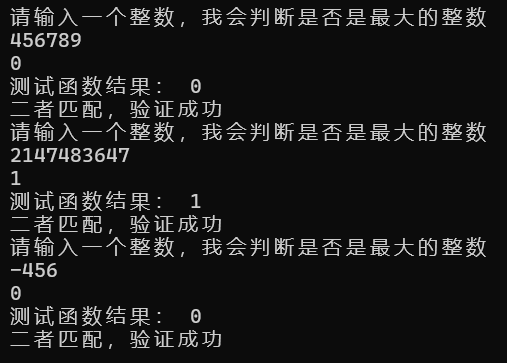
****

****

**int isTmax(int x)函数实现分析：**

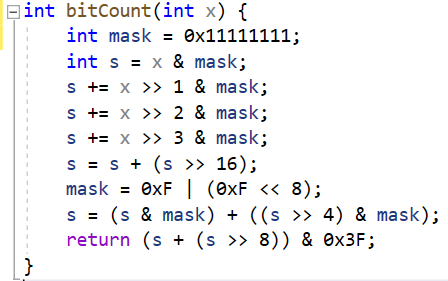
要判断一个数是不是最大的整数（int）可以利用这个数加上1<<31,取反后就是0，再用一个！就变为一个非零数。

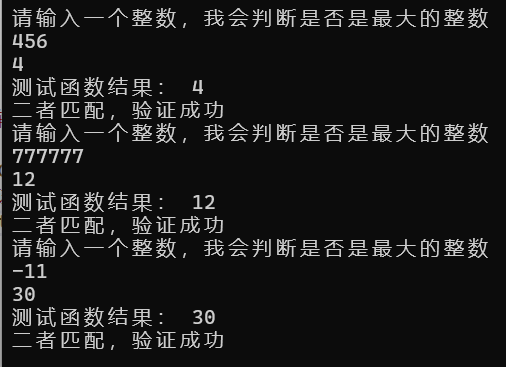
****

****

**int bitCount(int x)函数实现分析：**

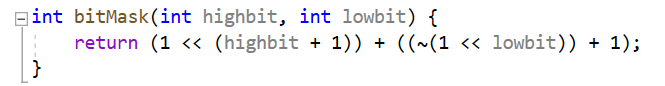
一般情况下我们习惯用while循环或者for循环来计算而二进制表示中1的个数，但是如果不能用这些工具的情况下我们可以先构造出0x11111111，以四个bit为单位，依次和传进来的参数进行&运算，这样每个四位bit显示的数组就是这四个bit中1的个数，接下来我们需要考虑的就是如何相加起来，采用向右移16位直接相加，这样我们就得到16位记录着1的个数的一个二进制表示，最后我们再构造0x0101，依次取出低第一个和第三个，低第二个和第四个的1的个数，最后再相加就可以了。

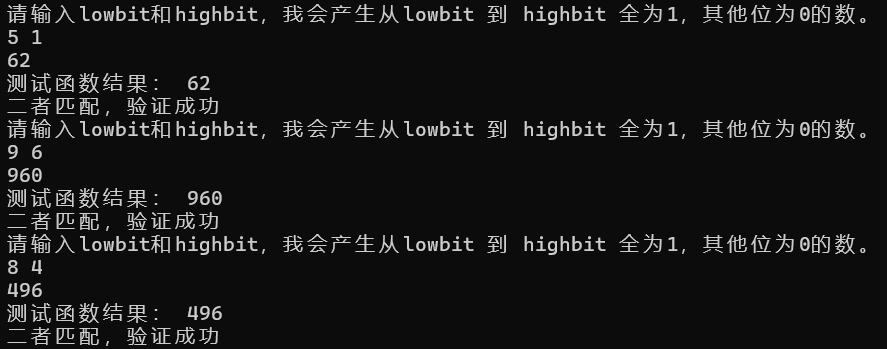
****

****

**int bitMask(int highbit, int lowbit)函数分析：**

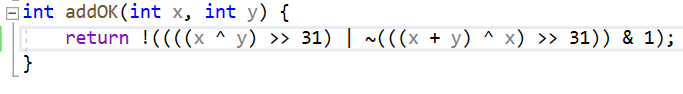
这个思路很妙，如果我想得到11100b，我可以用（100000-10）b。

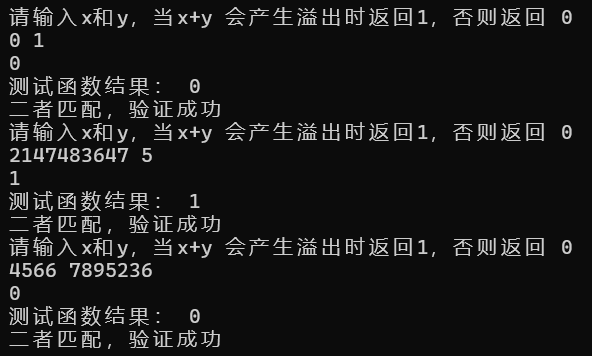
****

****

**int addOK(int x, int y)函数实现分析：**

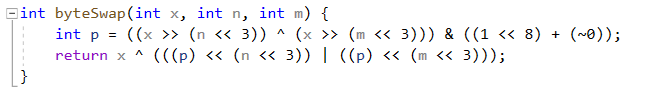
对于两个异号的数相加，一定不会溢出，我们只需要考虑两个同号数字的相加，我们可以用(x ^ y) >> 31判断是否异号，用~(((x + y) ^ x) >> 31)来判断如果x和y同号的话溢出情况，如果没溢出就是一个非零的数，我们最后再将两种情况|后再&1即可。

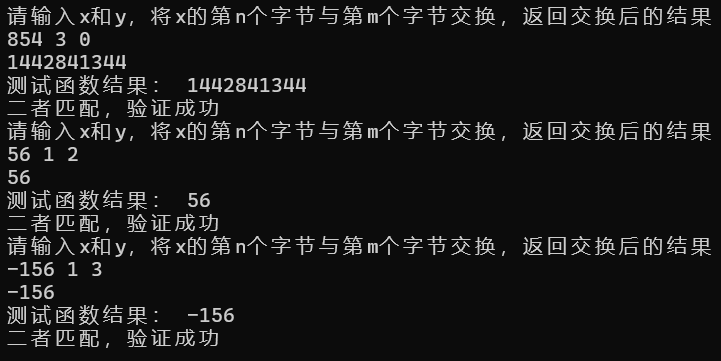
****

****

**int byteSwap(int x, int n, int m)函数实现分析：**

首先我们需要提取字节x >> (n << 3)：将 x 右移 n \* 8 位，以将第 n 个字节移到最低字节位置。x >> (m << 3)：同样，将 x 右移 m \* 8 位，以将第 m 个字节移到最低字节位置。使用^将这两个字节合并，存储在变量 p 中,使用(1 << 8) – 1构造0xff，帮助提取单个字节，最后再使用|和^进行位置交换得到最后的答案。





**四、体会**

实践出真知，在实验中巩固和学习了很多知识，比如指针的使用，指针的强制类型转换使用，以及学习了解了数据在内存中的存储，还有就是巩固了大一学c的结构体的知识的时候提到的内存对齐的问题，像是遇见了老朋友一样。

对于任务二的话，长了我的见识的同时学会了从逻辑电路与门、或门、非门等等角度，实现CPU的常见功能，我好像进一步了解了数据在计算机内部的运行方式，受益匪浅。

**五、源码**

任务一源代码：

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<bits/stdc++.h>

#include<windows.h>

using namespace std;

struct student;

// 压缩函数

int pack\_student\_bytebybyte(student\* s, int sno, char\* buf);

int pack\_student\_whole(student\* s, int sno, char\* buf);

// 解压函数

int restore\_student(char\* buf, int len, student\* s);

// 输出函数

void printMessage(char\* message, int len);

// 延迟函数

void ddelay();

const int N = 5;

struct student

{

char name[8];

long long age;

float score;

char remark[196];

}new\_s[N];

int main() {

// 存放压缩的信息

char message[500];

//memset(message, 0, sizeof message);

// 初始化

student old\_s[N] = {

{"王国豪",15,43,"人生若只如初见"},

{"啦啦",2,98,"何事秋风悲画扇"},

{"笨蛋",3,96,"等闲变却故人心"},

{"什么",4,94,"却道故人心易变"},

{"鲁尼在",5,92,"沧海桑田"}

};

cout << "开始输出未压缩的内容......" << endl;

// 打印 old\_s[N]

for (int i = 0; i < N; i++) {

cout << old\_s[i].name << " " << old\_s[i].age << " " << old\_s[i].score << "　" << old\_s[i].remark << endl;

}

cout << "压缩前存放数据的长度为 :" << sizeof old\_s << endl;

cout << "下面开始按字节压缩前2条记录" << endl;

//ddelay();

// 按字节压缩

int len = pack\_student\_bytebybyte(old\_s, 2, message);

cout << "此时message的长度为: " << len << endl;

cout << "下面开始按一条一条压缩下3条记录" << endl;

//ddelay();

// 一条一条压缩

len += pack\_student\_whole(&old\_s[2], 3, message + len);

cout << "此时message的长度为: " << len << endl;

// 开始输出存储压缩信息的message中的内容

cout << "开始输出两次压缩后message中的内容" << endl;

printMessage(message, len);

cout << "下面开始解压message中的信息到新结构体中" << endl;

//ddelay();

int num = restore\_student(message, len, new\_s);

// 打印 new\_s[N]

for (int i = 0; i < num; i++) {

cout << new\_s[i].name << " " << new\_s[i].age << " " << new\_s[i].score << "　" << new\_s[i].remark << endl;

}

char\* pp = message;

cout << "以十六进制的形式，输出message的前20个字节的内容 " << endl;

for (int i = 0; i < 20; i++) {

printf("%02X ", (unsigned char)(\*(pp + i)));

}

return 0;

}

void printMessage(char\* message, int len) {

int cnt = 0;

char\* p = message;

while (p - message < len)

{

cout << p << " "; // 名字

p += strlen(p) + 1;

cout << \*((short\*)p) << " "; // 年龄

p += 2;

cout << \*((float\*)p) << " ";

p += 4;

cout << p << " " << endl;

p += strlen(p) + 1;

}

}

// s为待压缩数组的起始地址； sno 为压缩人数；

// buf 为压缩存储区的首地址；两个函数的返回均是调用函数压缩后的字节数

int pack\_student\_bytebybyte(student\* s, int sno, char\* buf) {

int cnts = 0;

int cntname, cntage, cntscore, cntremark, cntbuf = 0;

char\* p = (char\*)s;

char\* pp = buf;

while (cnts < sno)

{

// 读取名字

cntname = 0;

while (cntname < 8) {

if (\*p) { // 遇到\0

\*pp = \*p;

cntname++, cntbuf++;

p++, pp++;

}

else {

\*pp = '\0';

cntbuf++;

p += (8 - cntname);

pp++;

break;

}

}

// 读取年龄

cntage = 0;

while (cntage < 2)

{

\*pp = \*p;

cntbuf++, cntage++, p++, pp++;

}

p += 2;

// 读入float,占 4 个字节

cntscore = 0;

while (cntscore < 4)

{

\*pp = \*p;

cntbuf++, cntscore++, p++, pp++;

}

// 读取remark数组

cntremark = 0;

while (cntremark < 200)

{

if (\*p) {

\*pp = \*p;

cntbuf++, cntremark++, p++, pp++;

}

else {

\*pp = 0;

cntbuf++, pp++;

p += (200 - cntremark);

break;

}

}

cnts++;

}

return cntbuf;

}

// 按

int pack\_student\_whole(student\* s, int sno, char\* buf) {

int cnts = 0;

char\* p = (char\*)s;

char\* pp = buf;

student\* ppp = s;

while (cnts < sno)

{

// 拷贝名字

strcpy(pp, ppp[cnts].name);

pp += strlen(ppp[cnts].name) + 1;

// 拷贝age

\*((short\*)pp) = ppp[cnts].age;

pp += 2;

// 拷贝score

\*((float\*)pp) = ppp[cnts].score;

pp += 4;

// 拷贝备注

strcpy(pp, ppp[cnts].remark);

pp += strlen(ppp[cnts].remark) + 1;

cnts++;

}

return pp - buf;

}

// 解压函数

//buf 为压缩区域存储区的首地址；len为buf中存放数据的长度；

// s为存放解压数据的结构数组的起始地址； 返回解压的人数。

int restore\_student(char\* buf, int len, student\* s) {

int cnt = 0; // 记录解压的人数

char\* p = buf;

student\* pp = s;

int record = 0;

while ((p - buf) < len)

{

// 解压名字

strcpy(pp[cnt].name, p);

p += strlen(pp[cnt].name) + 1;

// 解压年龄

pp[cnt].age = \*((short\*)p);

p += 2;

// 解压分数

pp[cnt].score = \*((float\*)p);

p += 4;

// 解压备注

strcpy(pp[cnt].remark, p);

p += strlen(pp[cnt].remark) + 1;

// 条数加一

cnt++;

}

return cnt;

}

void ddelay() {

for (int i = 0; i <= 4; i++) {

cout << "\*";

Sleep(1000);

}

cout << endl;

}

任务二源代码：

#include<iostream>

#include<vector>

using namespace std;

// 1.返回 x 的绝对值

int absVal(int x);

// 评定函数1

int absVal\_standard(int x) { return (x < 0) ? -x : x; };

// 2.不使用负号，实现 -x

int negate(int x);

// 评定函数2

int negate\_standard(int x) { return -x; }

// 3.仅使用 ~ 和 |，实现 &

int bitAnd(int x, int y);

// 评定函数3

int bitAnd\_standard(int x, int y) { return x & y; }

// 4.仅使用 ~ 和 &，实现 |

int bitOr(int x, int y);

// 评定函数4

int bitOr\_standard(int x, int y) { return x | y; }

// 5.仅使用 ~ 和 &，实现 ^

int bitXor(int x, int y);

// 评定函数5

int bitXor\_standard(int x, int y) { return x ^ y; }

// 6.判断x是否为最大的正整数（7FFFFFFF）

int isTmax(int x);

// 评定函数6

int isTmax\_standard(int x) { return x == 0x7fffffff; }

// 7.统计x的二进制表示中 1 的个数

int bitCount(int x);

// 评定函数7

int bitCount\_standard(int x) {

int a = 0;

for (int i = 0; i < 32; i++) {

if (((x >> i) & 1)) {

a++;

}

}

return a;

}

// 8.产生从lowbit 到 highbit 全为1，其他位为0的数

int bitMask(int highbit, int lowbit);

// 评定函数8

int bitMask\_standard(int highbit, int lowbit) {

int ans = 0;

for (int i = lowbit; i <= highbit; i++) {

ans += (1 << (i));

}

return ans;

}

// 9.当x+y 会产生溢出时返回1，否则返回 0

int addOK(int x, int y);

// 评定函数9

int addOK\_standard(int x, int y) {

long long lsum = (long long)x + y;

return !(lsum == (int)lsum);

}

// 10.将x的第n个字节与第m个字节交换，返回交换后的结果。

int byteSwap(int x, int n, int m);

// 评定函数10

int byteSwap\_standard(int x, int n, int m) {

unsigned int nmask, mmask;

switch (n) {

case 0:

nmask = x & 0xFF;

x &= 0xFFFFFF00;

break;

case 1:

nmask = (x & 0xFF00) >> 8;

x &= 0xFFFF00FF;

break;

case 2:

nmask = (x & 0xFF0000) >> 16;

x &= 0xFF00FFFF;

break;

default:

nmask = ((unsigned int)(x & 0xFF000000)) >> 24;

x &= 0x00FFFFFF;

break;

}

switch (m) {

case 0:

mmask = x & 0xFF;

x &= 0xFFFFFF00;

break;

case 1:

mmask = (x & 0xFF00) >> 8;

x &= 0xFFFF00FF;

break;

case 2:

mmask = (x & 0xFF0000) >> 16;

x &= 0xFF00FFFF;

break;

default:

mmask = ((unsigned int)(x & 0xFF000000)) >> 24;

x &= 0x00FFFFFF;

break;

}

nmask <<= 8 \* m;

mmask <<= 8 \* n;

return x | nmask | mmask;

}

int main() {

int n,m,u;

int t = 3;

while (t--)

{

cout << "请输入x和y，将x的第n个字节与第m个字节交换，返回交换后的结果" << endl;

cin >> n >> m >> u;

cout << byteSwap(n,m,u) << endl;

cout << "测试函数结果： " << byteSwap\_standard(n,m,u) << endl;

if (byteSwap(n,m,u) == byteSwap\_standard(n,m,u)) {

cout << "二者匹配，验证成功" << endl;

}

else {

cout << "验证失败" << endl;

}

}

return 0;

}

// 遇到的问题：

int absVal(int x) {

return (x + (x >> 31)) ^ (x >> 31);

}

// 直接取反加一

int negate(int x) {

return (~x)+1;

}

// 一开始想的是用逻辑电路里面的 ~(x | y),但是发现行不通

// & 有0则为false，则相当于 与门

// | 有1则为true,则相当于 或门

// 直接两次取反就可以

int bitAnd(int x, int y) {

return ~((~x) | (~y));

}

// 直接借鉴上面的

int bitOr(int x, int y) {

return ~((~x) & (~y));

}

// 借鉴数字电路中的异或门实现

int bitXor(int x, int y) {

return ~((~((~x) & y)) & (~(x & (~y))));

}

// -1 的取反是 0

int isTmax(int x) {

return !(~(x+(1<<31)));

}

// 巧妙使用

int bitCount(int x) {

int mask = 0x11111111;

int s = x & mask;

s += x >> 1 & mask;

s += x >> 2 & mask;

s += x >> 3 & mask;

s = s + (s >> 16);

mask = 0xF | (0xF << 8);

s = (s & mask) + ((s >> 4) & mask);

return (s + (s >> 8)) & 0x3F;

}

// 10000 - 100

int bitMask(int highbit, int lowbit) {

return (1 << (highbit + 1)) + ((~(1 << lowbit)) + 1);

}

// 如果异号的话前面为 1 ，直接为真

// 如果异号的情况下，((x ^ y) >> 31) 为0，对后续不影响,

// ~(((x + y) ^ x) >> 31) 这个判断计算出来的是否和之前的同号

int addOK(int x, int y) {

return !((((x ^ y) >> 31) | ~(((x + y) ^ x) >> 31)) & 1);

}

int byteSwap(int x, int n, int m) {

int p = ((x >> (n << 3)) ^ (x >> (m << 3))) & ((1 << 8) + (~0));

return x ^ (((p) << (n << 3)) | ((p) << (m << 3)));

}