

**课 程 实 验 报 告**

**课程名称： 计算机系统基础实验**

**实验名称： 数据的表示**

**院 系 ：计算机科学与技术**

**专业班级 ： CS2201**

**学 号 ： U20225365**

**姓 名 ： 叶俊江**

**指导教师 ： 朱虹**

**2024 年 3 月 22 日**

**一、实验目的与要求**

⑴ 熟练掌握程序开发平台(VS2019) 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);

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。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个部分的编码，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 编写位运算程序**

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：**

任务中需要压缩的结构体数组的结构如下：

图形用户界面, 文本, 应用程序

描述已自动生成

图 1-1 结构体数组student的结构

那么任务描述中的压缩与解压就可以理解为：将old\_s[N]中的name以及remark数组中多余的空间以及short类型的age与float类型的score之间空余的2个字节（对齐）的多余空间去掉，存储于char类型message数组中，解压则将message数组的中的数组读取到新的结构体数组new\_s[N]中。

以下为任务实现代码中关键函数的分析：

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

此为压缩函数，并且以逐个字节的形式压缩，其中s为压缩数组的始地址,sno为压缩人数,buf为压缩存储区的首地址,返回值为压缩后的字节数。函数的主要思想为：通过char\*类型的指针p来遍历结构体成员的内存，并且将不为空的数据复制到message的内存中，值得注意的是，遍历字符串的时候遇到’\0’的时候停止遍历，然后指针加上相应的偏移量来进行下一步压缩。

图标

描述已自动生成 文本

描述已自动生成

图1-2 输入的old\_s[N]结构体数组数据 图1-3 输出的message数据以及两者内存大小

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

此同为压缩函数，但是此函数采用逐条信息的形式压缩，其中s为压缩数组的始地址,sno为压缩人数,buf为压缩存储区的首地址,返回值为压缩后的字节数。因为任务书中要求对short、float字段只能用一条语句整体写入，则使用strcpy函数拷贝字符串，快速接收字符。采用指针类型强制转换来拷贝old\_s[N]中的short和float变量：先定义一个student类型的指针pp来接受我们需要压缩的结构体数组，则pd[cnt].age，pd[cnt].score便定位到这两个short和float数据的位置，并且注意类型转换的表达：\*((short \*)pd) = pp[cnt].age、\*((float \*)pd) = pp[cnt].score，

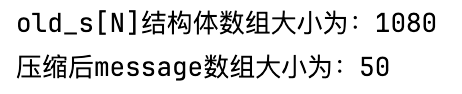


图1-4 压缩前后old\_s[N]与message两者的内存大小

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

此函数为解压函数，其中buf为压缩存储区的首地址，len为压缩数据段长度，s为解压结构数组首地址，返回解压人数。与上述pack\_student\_whole函数类似，函数主要采用了strcpy函数拷贝字符串，同时利用指针类型强制转换来拷贝short和float变量，拷贝时注意short和float内存对齐的多余两个字节的空间。

**压缩及解压全过程调试分析：**

通过调试可以查询到old\_s[N]，message以及new\_s[N]的地址以及他们的内存信息。调试使用的输入示例为：

图片包含 图标

描述已自动生成

图1-5 调试输入示例

表格

描述已自动生成

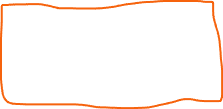


图1-6 old\_s[N]结构体数组的内存信息（部分）

表格

描述已自动生成



图1-7 message数组的内存信息

上图6、图7中，红色圆圈圈起的数据为name，蓝色圆圈为age，绿色圆圈为score，橙色圆圈为remark。很明显地看出压缩前很多内存空间未得到有效利用，压缩后占用内存明显减少。在圈起的数据中信息均一致，说明压缩过程正常。

图形用户界面, 应用程序, 表格, Excel

描述已自动生成



图1-8 new\_s[N]结构体数组的内存信息（部分）

上图图8为解压后的new\_s[N]结构体数组的内存信息，对比前两图信息正确，说明解压过程正常。

全过程运行结果截图如下：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图1-9全过程运行结果截图

**message前20个字节的内容分析验证：**

以十六进制的形式，输出message的前20个字节的内容的执行结果如下图：



图1-10 message的前20个字节的内容

在内存视图中，message的前20个字节的内容为：

图形用户界面, 文本, 应用程序

描述已自动生成

图 1-11 内存视图中message的前20个字节的内容

对比可知，两者内容一致，验证完毕。

**对于第0个学生的score验证浮点数编码:**

由上图图5可知输入示例中第0个学生的score数据为24，按照单精度浮点数编码可得:

文本

描述已自动生成

经查询，得内存视图中score数据为：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成



图 1-12 内存视图中第0个学生的score内容

上图图11中红色圆圈中即为所要score内容，并且注意高字节存于高地址，低字节存于低地址。验证完毕。

**结构数组中个元素的存放规律：**

由以上分析可知，结构体变量的首地址为其最长基本类型成员的整数倍，并且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，若有需要编译器会对其做出一些调整，例如old\_s[N]结构体中short和float之间为了内存对齐添加了两个字节的空间。

**（2）任务 2 的算法思想、运行结果等记录**

**int absVal(int x)函数分析:**

求绝对值关键在于x的符号位，那么我们可以通过x>>31来获取最高位符号位的信息。而且需要注意的是，算术右移的时候，若x是负数，那么x>>31后得到的数为0xFFFFFFFF,则可以先减一后取反得到正数；若x是正数，那么x>>31后得到的数为0，对后续运算无影响。

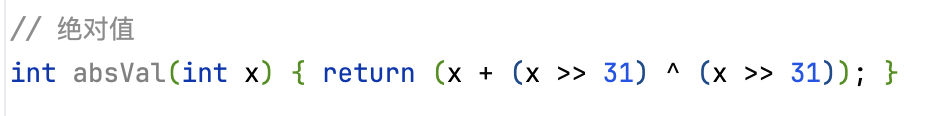


图2-1 int absVal(int x)函数体

**图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成**

图2-2 absVal函数验证运行结果截图

**int negate(int x)函数分析：**

由补码的知识可知：对操作数取反加一即可得到取负值的函数实现。

图片包含 徽标

描述已自动生成

图2-3 int negate(int x)函数体

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图2-4 negate函数验证运行结果截图

**int bitAnd(int x, int y)函数分析：**

参考数电逻辑中的与或门，则x&y = ~((~x)|(~y))。

徽标, 公司名称

描述已自动生成

图2-5 int bitAnd(int x, int y)函数体

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图2-6 bitAnd函数验证运行结果截图

**int bitOr(int x, int y)函数分析：**

与bitAnd函数思路类似，x|y = ~((~x)&(~y))。

徽标, 公司名称

描述已自动生成

图2-7 int bitOr(int x, int y)函数体



图2-8 bitOr函数验证运行结果截图

**int bitXor(int x, int y)函数分析：**

与bitAnd函数、bitOr函数思路类似，x^y = ~(~x & ~y) & ~(x & y)。

徽标

描述已自动生成

图2-9 int bitXor(int x, int y)函数体

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图2-10 bitXor函数验证运行结果截图

**int isTmax(int x)函数分析：**

所要判断的最大正整数为0x7FFFFFFF(int)，可以用这个数加上1<<31，即0x80000000，取反后等于0，这时候再取一个非便能得到非零的数，其他情况则得到0。

图片包含 徽标

描述已自动生成

图2-11 int isTmax(int x)函数体

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图 2-12 isTmax函数验证运行结果截图

**int bitCount(int x)函数分析：**

此函数的难点在于不能使用循环进行计算，若使用循环则运算次数会超出任务书所限定的范围。因而我们应该考虑构造出0x11111111，以四个bit为单位分别各自来运算bit中1的个数，随后右移16位相加，得到记录1的个数的二进制表示。最后构造0x0101来取出一、三位，二、四位的1的个数，随后相加即可。

图形用户界面, 文本

描述已自动生成

图 2-13 int bitCount(int x)函数体

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图2-14 bitCount函数验证运行结果截图

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

2<<highbit是让0x00000010右移highbit位，加上~0(0x11111111)后，得到从最高位到highbit+1位均为0，highbit到最低位均为1的二进制数；类似地，1<<lowbit+(~0)得到最高位到lowbit位均为0，lowbit-1位到最低位均为1的二进制数，随后取反得最高位到lowbit位均为1，lowbit-1位到最低位均为0的二进制数，随后两者做&运算得到。

图片包含 图形用户界面

描述已自动生成

图2-15 int bitmask(int highbit, int loewbit)函数体



图2-16 bitMask函数验证运行结果截图

**int addOK(int x, int y)函数分析：**

对x，y以及他们的和sum三个数取出他们的符号位并且右移至最低位，若x+y会溢出，则x与y符号相同，sum与x符号不同；反之，x与y符号不同或者x与sum符号相同，则没发生溢出。取两次！以便于非溢出时得到返回值0。

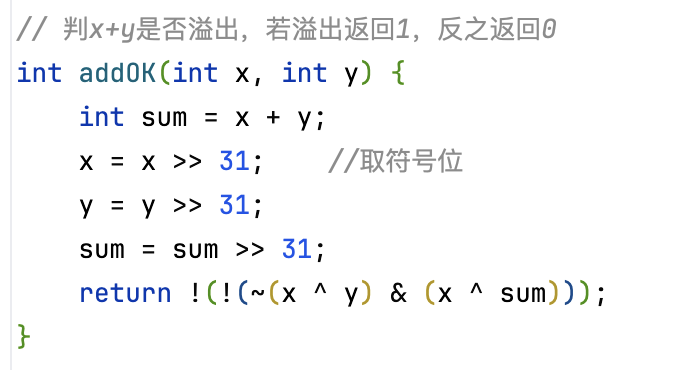


图2-17 int addOK(int x, int y)函数体

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图2-18 addOK函数验证运行结果截图

**int byteSwap(int x, int n, int m)函数分析：**

将n、m左移3位得到位索引nTemp、mTemp，nNum、mNum是方便后续得到nByte、mByte而创建的掩码，nByte和mByte则为x中的第n个字节和第m个字节，而byteNum为除了n、m字节外其他各位均为1的数，over则是x与byteNum通过和运算得到x除了n、m字节为0外的数，此时将nByte移到第m个字节，mByte移到第n个字节再与over做或运算即可得到。

图形用户界面, 文本, 应用程序

描述已自动生成

图2-19 int byteSwap(int x, int n, int m)函数体

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图2-20 byteSwap函数验证运行结果截图

**四、体会**

通过完成本次实验任务，我对于结构体的处理、内存布局以及位运算等方面有了更深入的理解。在任务1中，我学会了如何压缩和解压结构体数组，并且了解了内存对齐的原理和处理方法；而在任务2中，通过位运算的实现，我加深了对于计算机底层操作的认识，提高了编程能力。在实验中，不仅需要编写代码，还需要通过调试来验证程序的正确性。通过调试过程，我学会了如何利用调试工具观察内存数据、变量值等信息，帮助我理解程序的执行过程，找出问题并进行修复。

实验要求中提到了一些特定的算法要求，如绝对值的计算、位运算的实现等。通过实际编码、查询资料来实现这些算法，我对于这些算法的思想有了更清晰的认识，提升了我的算法设计能力。在实验中，对于函数的参数传递、数据类型的转换、内存布局等细节都需要仔细考虑和处理，这提高了自己对于编程规范性的要求。

通过这次实验，我将课堂上学到的理论知识应用到实际编码中，加深了对于课程内容的理解。同时，实验中遇到的问题也促使我深入思考和学习，提高了我对于计算机原理和编程技术的综合应用能力。

**五、源码**

实验任务 1、2的源程序（单倍行距，5号宋体字）

任务1源程序\.task\_1.cpp

#include <iostream>

#include <cstring>

using namespace std;

const int N = 5;

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message函数

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

//十六进制输出函数

void printMessageBy0x(char \*message, int len);

// student结构

struct student {

char name[8];

short age;

float score;

char remark[200];

} new\_s[N];

int main() {

student old\_s[N];

for (auto &old\_: old\_s) {

cin >> old\_.name >> old\_.age >> old\_.score >> old\_.remark;

}

// 打印old\_s[N]

for (auto &old\_: old\_s) {

cout << old\_.name << " " << old\_.age << " " << old\_.score << " " << old\_.remark << endl;

}

// 初始化message

char message[500];

memset(message, 0, sizeof message);

// N1=2\按字节压缩

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

// N2=3\整体压缩

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

printMessage(message, len);

printf("old\_s[N]结构体数组大小为：%d\n", sizeof(old\_s));

printf("压缩后message数组大小为：%d\n", len);

printMessageBy0x(message, len);

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].score << " "

<< new\_s[i].remark << endl;

}

return 0;

}

// 按字节压缩

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

// s为压缩数组的始地址,sno为压缩人数,buf为压缩存储区的首地址,返回值为压缩后的字节数

int cnt = 0; // 计数器

int cntName, cntAge, cntScore, cntRemark, cntBuf = 0;

char \*p = (char \*) s; // 指向待操作数组

char \*pd = buf; // 指向压缩存储地址

while (cnt < sno) {

// 读取name

cntName = 0;

while (cntName < 8) {

if (\*p) {

// p不为空时

\*pd = \*p;

cntName++, cntBuf++;

p++, pd++;

} else {

//\*p为\0时

\*pd = 0;

cntBuf++;

p += (8 - cntName); // name可能存在空余

pd++;

break;

}

}

// 读取age

cntAge = 0;

while (cntAge < 2) {

\*pd = \*p;

cntBuf++, cntAge++;

p++, pd++;

}

p += 2; // 内存空间中short与float之间有2个字节的未利用空间

// 读取score

cntScore = 0;

while (cntScore < 4) {

\*pd = \*p;

cntBuf++, cntScore++;

pd++, p++;

}

// 读取remark

cntRemark = 0;

while (cntRemark < 200) {

if (\*p) {

\*pd = \*p;

cntBuf++, cntRemark++;

pd++, p++;

} else {

\*pd = 0;

cntBuf++, pd++;

p += (200 - cntRemark); // 空余处理

break;

}

}

cnt++;

}

return cntBuf;

}

// 整体压缩

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

int cnt = 0;

char \*pd = buf;

student \*pp = s;

while (cnt < sno) {

// copy name

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

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

// copy age

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

pd += 2;

// copy score

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

pd += 4;

// copy remark

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

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

cnt++;

}

return pd - buf; // 末地址减始地址

}

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

char \*p = message;

while (p - message < len) {

cout << p << " "; // name

p += strlen(p) + 1;

cout << \*((short \*) p) << " "; // age

p += 2;

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

p += 4;

cout << p << " "; // remark

p += strlen(p) + 1;

}

cout << endl;

}

// 解压函数

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

// buf为压缩存储区的首地址，len为压缩数据段长度，s为解压结构数组首地址，返回解压人数

int cnt = 0; // 计数器->人数

char \*p = buf;

student \*pd = s;

while ((p - buf) < len) {

// restore name

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

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

// restore age

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

p += 2;

// restore score

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

p += 4;

// restore remark

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

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

cnt++; // 处理完人数++

}

return cnt;

}

void printMessageBy0x(char \*message, int len) {

char \*p = message;

int bytesToOutput = 20; // 只输出前20个字节的内容

for (int i = 0; i < len && i < bytesToOutput; ++i) {

printf("%02X ", (unsigned char) p[i]);

}

cout << endl;

}

任务二源程序\.task\_2.cpp

#include <iostream>

// 绝对值

int absVal(int x) { return (x + (x >> 31) ^ (x >> 31)); }

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

// 负值

int negate(int x) { return (~x) + 1; }

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

// 和

int bitAnd(int x, int y) { return ~((~x) | (~y)); }

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

// 或

int bitOr(int x, int y) { return ~((~x) & (~y)); }

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

// 异或

int bitXor(int x, int y) { return ~(~x & ~y) & ~(x & y); }

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

// 判最大正整数(7FFFFFF)

int isTmax(int x) { return !(~(x + (1 << 31))); }

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

// 计算x二进制表示中1个数

int bitCount(int x) {

int n1 = 0x11 | (0x11 << 8); //0 0 0 0 1 1 1 1

int num = n1 | (n1 << 16); //1 1 1 1 1 1 1 1

int account = x & num;

account += x >> 1 & num;

account += x >> 2 & num;

account += x >> 3 & num; //分别对各个四位操作

account += (account >> 16);

num = 0xf | (0xf << 8);

account = (account & num) + ((account >> 4) & num);

return (account + (account >> 8)) & 0x3f;

}

int bitCount\_standard(int x) {

int ret = 0;

while (x) {

x &= x - 1;

ret++;

}

return ret;

}

// 产生从lb到hb全为1，其余位为0的数

int bitMask(int highbit, int lowbit) {

return ((2 << highbit) + (~0)) & ~((1 << lowbit) + (~0));

}

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

int result = 0;

int i;

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

result |= 1 << i;

return result;

}

// 判x+y是否溢出，若溢出返回1，反之返回0

int addOK(int x, int y) {

int sum = x + y;

x = x >> 31; //取符号位

y = y >> 31;

sum = sum >> 31;

return !(!(~(x ^ y) & (x ^ sum)));

}

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

if (x > 2147483647 - y) {

return 1;

} else if (x < -2147483648 - y) {

return 1;

} else {

return 0;

}

}

// 将x中第n个字节与第m个字节交换，返回交换后结果

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

int nTemp = n << 3;

int mTemp = m << 3;

int nNum = 0xff << nTemp;

int mNum = 0xff << mTemp;

int nByte = ((x & nNum) >> nTemp) & 0xff;

int mByte = ((x & mNum) >> mTemp) & 0xff;

int byteNum = nNum | mNum;

int over = x & ~byteNum;

return over | (nByte << mTemp) | (mByte << nTemp);

}

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() {

//测试函数，根据需要修改main（）函数

int x, y, z;

std::cin >> x >> y >> z;

while (true) {

// int x;

// std::cin >> x;

// while (x) {

std::cout << "byteSwap(x)函数值为：" << byteSwap(x, y, z) << std::endl;

std::cout << "byteSwap\_standard(x)函数值为：" << byteSwap\_standard(x, y, z) << std::endl;

if (byteSwap\_standard(x, y, z) == byteSwap(x, y, z)) {

std::cout << "byteSwap()函数与预期一致！" << std::endl;

}

if (x == 0 && y == 0 && z == 0) break;

std::cin >> x >> y >> z;

// std::cin >> x;

}

return 0;

}