

华中科技大学

# 课程实验报告

课程名称： 计算机系统基础

实验名称： 数据表示和等效运算

院 系： 计算机科学与技术

专业班级： CS2306

学 号：                     

姓 名：                     

指导教师：                     

2025 年 3 月 13 日

## 一、实验目的与要求

- (1) 熟练掌握程序开发平台(VS2019/VS2022) 的基本用法, 包括程序的编译、链接和调试;
- (2) 熟悉数据的表示形式;
- (3) 熟悉地址的计算方法、地址的内存转换;
- (4) 用常见的按位操作(移位、按位与/或/非/异或) 等实现运算表达式的等效运算。

## 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, 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。

- (3) 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 40 个字节的内容, 并与调试时在内存窗口观察到的 message 的前 40 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; }`
- (2) `int negate(int x);` 不使用负号，实现 `-x`  
判断函数：`int negate_standard(int x) { return -x; }`
- (3) `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; }`
- (4) `int bitOr(int x, int y);` 仅使用 `~` 和 `&`，实现 `|`
- (5) `int bitXor(int x, int y);` 仅使用 `~` 和 `&`，实现 `^`
- (6) `int isTmax(int x);` 判断 `x` 是否为最大的正整数（`7FFFFFFF`），  
只能使用 `!`、`~`、`&`、`^`、`|`、`+`
- (7) `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
- (8) `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。
- (9) `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
- (10) `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
- (11) `int bang(int x)` 当 `x=0` 时，返回 1；其他情况返回 0。实现逻辑非(!)  
例：`bang(3)=0; bang(0)=1;`  
仅使用 `~`、`&`、`^`、`|`、`+`、`<<`、`>>`，运算次数不超过 12 次  
提示：只有当 `x=0` 时，`x` 与 `-x` 的最高二进制位会同时为 0。
- (12) `int bitParity(int x)` 当 `x` 有奇数个二进制位 0，返回 1；否则返回 0  
例：`bitParity(5)=0; bitParity(7)=1;`  
仅使用 `!`、`~`、`&`、`^`、`|`、`+`、`<<`、`>>`，运算次数不超过 20 次。  
提示：只有当 `x` 的高字与低字的对应位（第 0 位对应第 16 位，第 1 位对应第 17 位，依次类推）同时为 1，则出现了成对的二进制位 1，此时，可以将对应的二进制位置为 0，不会影响二进制位 1 的个数的奇偶性判断。

### 三、实验记录及问题回答

#### (1) 任务 1 的算法思想、运行结果、观察记录问题的解答等记录

##### 1. 分析与观察

首先，对待压缩的数据进行分析。

对于结构体 `student`，姓名 `name` 分配了 8 个字节，而名字的长度不会总是占 8 个字节，因此，`name` 数据具有压缩的必要。其次是年龄 `age`，占据 2 个字节，`short` 数据类型的范围是 -32768~32767，而学生的年龄最低不小于 0，最高也不会超过 100，因此单字节足以表示年龄。之后是分数 `score`，数据类型为浮点型，由于题目未说明分数的具体范围和格式，如最多会有几位小数，因此考虑最复杂的情况，在最复杂的情况下，根据 IEEE 754 标准，`float` 所占的 4 个字节没有压缩的空间。最后，是评价 `remark`，可见分配了 200 个字节，有很大的压缩余地。综上所述，压缩的主体在两个 `char` 数组，以及年龄的一个字节。

在 x86 环境下，可以通过 vs2022 工具观察到结构体在内存中的具体结构。

```
彭冲 18 96 Excellent
```

```
Jerry 19 88.5 Good
```

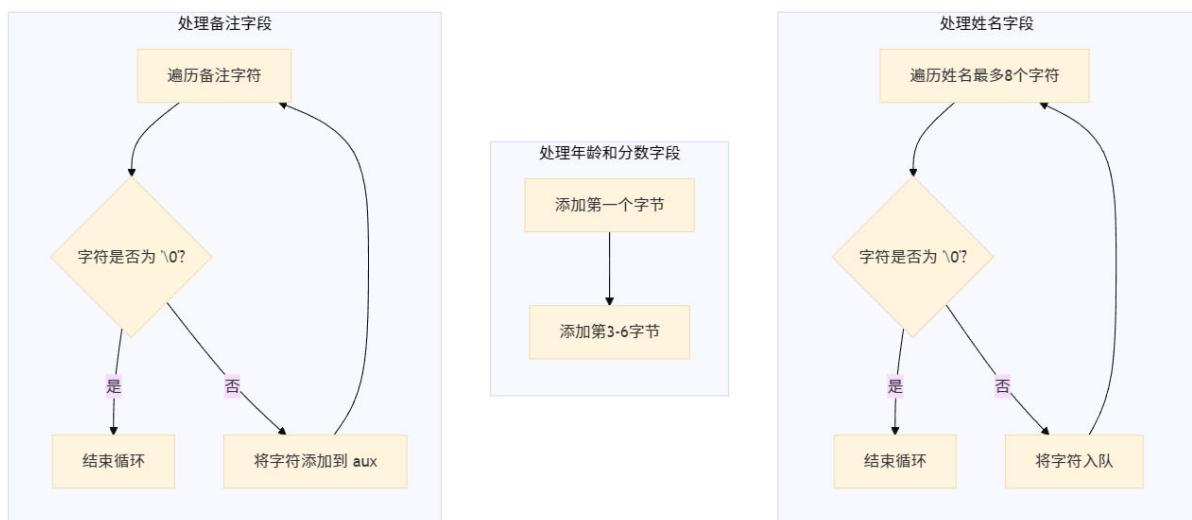
```
Rose 20 77.5 Fair
```

```
Jack 21 66.5 Poor
```

```
Lily 22 55.5 Fail
```

当学生数据为以上信息时，内存中可以看到，除了红色的有效信息外，大部分内存字节均为连续的白色 00，表明其属于冗余存储，具有压缩的空间，与前面所述的分析相符。

## 2. 算法思想与流程



对于 `size = sizeof(student) * student num` 的连续内存，使用字符指针 `char* p` 对该

内存进行遍历。

首先，逐字节读取姓名，直到遇到 '\0' 或达到 8 个字符停止，添加一个 '\0' 作为名字的结束标志。

然后，处理年龄和分数字段的 6 个字节，直接添加第一个字节(年龄的低字节)和第 3-6 字节(分数的 4 个字节)，并跳过年龄的高字节(第 2 个字节)。

最后，备注字段的 200 个字节的处理与姓名相同。

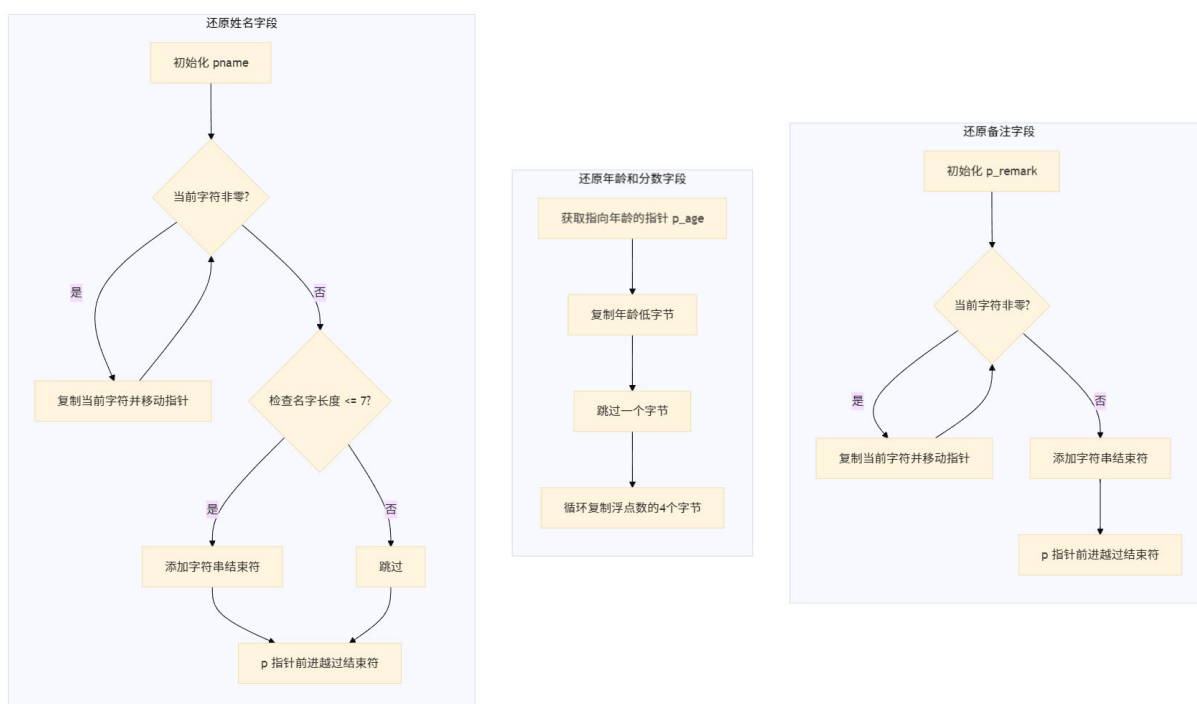
接着，递增指针直到遇到非零字节，即为另一个 student 的 name 字段，重复上述内容；或者，指针超过 student 数组的界限，那么便停止遍历。

## 2.2 函数 pack\_student\_whole

与 pack\_student\_bytebybyte 类似，但借助 student 结构体的信息，使用 strcpy 和 memcpy 直接对变量进行复制。

## 2.3 函数 restore\_student

即为压缩过程的逆向操作。



### 3. 运行结果

```
Please input 5 students' information(name, age, score, remark), sepearte with space:
彭冲 18 96 Excellent
Jerry 19 88.5 Good
Rose 20 77.5 Fair
Jack 21 66.5 Poor
Lily 22 55.5 Fail

Scanned 5 students' information
Raw size: 1070 bytes.

Compressed size: 84 bytes.
Compress ratio: 7.85%

name      age score remark
彭冲      18  96.0 Excellent
Jerry     19  88.5 Good
Rose      20  77.5 Fair
Jack      21  66.5 Poor
Lily      22  55.5 Fail
```

可见，压缩效果显著，压缩后体积仅为原本的 7.85%。压缩后内存中数据如下图所示，

可见数据紧凑。

```
0x012C6080 c5 ed b3 e5 00 12 00 00 c0 42 45 78 63 65 6c 6c  ????.???BExcell
0x012C6090 65 6e 74 00 4a 65 72 72 79 00 13 00 00 b1 42 47  ent.Jerry....?BG
0x012C60A0 6f 6f 64 00 fd fd fd fd a9 44 69 e4 00 02 00 80  ood.?????Di? ... €

0x012BD7D0 52 6f 73 65 00 14 00 00 9b 42 46 61 69 72 00 4a  Rose....?BFair.J
0x012BD7E0 61 63 6b 00 15 00 00 85 42 50 6f 6f 72 00 4c 69  ack....?BPoor.Li
0x012BD7F0 6c 79 00 16 00 00 5e 42 46 61 69 6c 00 cd cd cd  ly....^BFail.???
```

#### (2) 任务 2 的各个等效运算的算法思想、运行结果等记录

算法思想：

##### 1. absVal(int x) - 求绝对值

算法思想：若  $x \geq 0$ ， $|x| = x \wedge 0x00000000 + 0$ ；若  $x < 0$ ， $|x| = x \wedge 0xFFFFFFFF + 1$

- 1) 获取  $x$  的符号位：通过  $x \gg 31$  得到全 0（正数）或全 1（负数）的 mask。
- 2) 若  $x$  为正数，进行  $(x \wedge 0) + 0$  不改变值。
- 3) 若  $x$  为负数，进行  $(x \wedge 0xFFFFFFFF) + 1$  等价于取反加一，实现了负数变正数。

2. `negate(int x)` - 不使用负号实现  $-x$

算法思想：使用位级补码表示的基本原则，取反加一： $\sim x + 1$  即得到  $-x$ 。

3. `bitAnd(int x, int y)` - 用  $\sim$  和  $|$  实现  $\&$

算法思想：应用德摩根定律： $x \& y = \sim(\sim x | \sim y)$ ，先对  $x$  和  $y$  分别取反，进行或运算，再对结果取反。

4. `bitOr(int x, int y)` - 用  $\sim$  和  $\&$  实现  $|$

算法思想：应用德摩根定律： $x | y = \sim(\sim x \& \sim y)$ ，先对  $x$  和  $y$  分别取反，进行与运算，再对结果取反。

5. `bitXor(int x, int y)` - 用  $\sim$  和  $\&$  实现  $\wedge$

算法思想：异或表示两个位不同时为 1，可以通过  $\sim(\sim(x \& \sim y) \& \sim(\sim x \& y))$  实现。  
 $(x \& \sim y)$  表示  $x$  为 1,  $y$  为 0 的位， $(\sim x \& y)$  表示  $x$  为 0,  $y$  为 1 的位，再通过德摩根定律构造异或。

6. `isTmax(int x)` - 判断是否为最大正整数

算法思想：直接检查  $x$  是否等于  $0x7FFFFFFF$ （最大正整数），使用  $!(x \wedge 0x7FFFFFFF)$  判断是否相等。

7. `bitCount(int x)` - 统计二进制表示中 1 的个数

算法思想：普通方法很容易超过运行次数限制。使用分治法，先将相邻的 1 位组累加，再将相邻的 2 位组累加，依此类推。每次累加使用掩码隔离需要计算的位，然后右移并累加，掩码依次为： $0x55555555$ ,  $0x33333333$ ,  $0x0F0F0F0F$ ,  $0x00FF00FF$ ,  $0x0000FFFF$ 。

8. `bitMask(int highbit, int lowbit)` - 生成指定范围的掩码

算法思想：创建一个从最低位到 `highbit` 位都是 1 的掩码，再创建一个从最低位到 `lowbit-1` 位都是 1 的掩码，两者相减，得到从 `lowbit` 到 `highbit` 位都是 1，其余位都是 0 的掩码，实现为  $(1 \ll (\text{highbit} + 1)) + \sim(1 \ll \text{lowbit}) + 1$



9. addOK(int x, int y) - 判断加法是否溢出

算法思想：溢出条件是两个正数相加得负数，或两个负数相加得正数。检查  $x$  和  $y$  是否同号： $((x \wedge y) \gg 31) + 1$ （同号为 1，异号为 0），再检查  $x$  和  $x+y$  是否同号： $((x + y) \wedge x) \gg 31) + 1$ （同号为 1，异号为 0），当  $x$  和  $y$  同号，且  $x$  和  $x+y$  异号时，说明发生了溢出。

10. byteSwap(int x, int n, int m) - 交换两个字节

算法思想：用掩码  $0xFF \ll n*8$  和  $0xFF \ll m*8$  提取  $n$  和  $m$  位置的字节，通过异或运算将对应的字节清零，之后将  $n$  字节位置的值移动到  $m$  字节的位置，将  $m$  字节位置的值移动到  $n$  字节的位置，将移动后的值与原  $x$ （已清除  $n$  和  $m$  字节）进行或运算。

11. bang(int x) - 实现逻辑非

算法思想：利用 0 与其他数的特殊性质：只有 0 的逻辑非为 1，其他都为 0。观察：只有  $x = 0$  时， $x$  和  $-x$  的最高位同时为 0，取出  $x$  和  $-x$  的符号位，进行或运算，再取反 $((x \gg 31) \& 1) \mid (((\sim x + 1) \gg 31) \& 1)) \wedge 1$ 。

12. bitParity(int x) - 判断二进制位 1 的奇偶性

算法思想：用分治法，通过异或运算折叠位。先将高 16 位与低 16 位异或，再将结果的高 8 位与低 8 位异或，依此类推，最终得到最低位的值，即为奇偶校验结果。

程序测试：

充分考虑参数各种情形，将函数与标准函数的结果进行比较，并使用 assert 断言判断结果是否相同。

```
void test() {
    assert(absVal(8) == absVal_standard(8), "absVal(+) failed\n");
    assert(absVal(-8) == absVal_standard(-8), "absVal(-) failed\n");

    assert(negate(8) == negate_standard(8), "negate(+) failed\n");
    assert(negate(-8) == negate_standard(-8), "negate(-) failed\n");

    assert(bitAnd(0x12345678, 0x87654321) == bitAnd_standard(0x12345678, 0x87654321), "bitAnd() failed\n");
    assert(bitOr(0x12345678, 0x87654321) == bitOr_standard(0x12345678, 0x87654321), "bitOr() failed\n");
    assert(bitXor(0x12345678, 0x87654321) == bitXor_standard(0x12345678, 0x87654321), "bitXor() failed\n");

    assert(isTmax(0x12345678) == isTmax_standard(0x12345678), "isTmax(0x12345678) failed\n");
    assert(isTmax(0x7FFFFFFF) == isTmax_standard(0x7FFFFFFF), "isTmax(0x7FFFFFFF) failed\n");

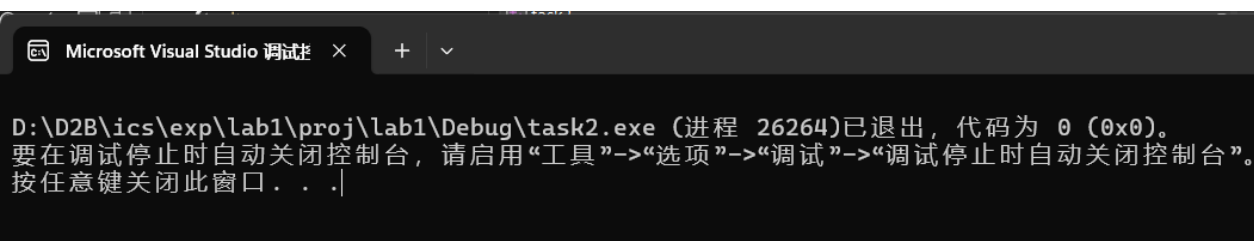
    assert(bitMask(5, 3) == bitMask_standard(5, 3), "bitMask(5, 3) failed\n");

    assert(addOK(0x12345678, 0x87654321) == addOK_standard(0x12345678, 0x87654321), "addOK(+,-) failed\n");
    assert(addOK(0x12345678, 0x7F654321) == addOK_standard(0x12345678, 0x7F654321), "addOK(+,+) failed\n");
    assert(addOK(0xFFFFFFFF, 0x80000000) == addOK_standard(0xFFFFFFFF, 0x80000000), "addOK(-,-) failed\n");

    assert(byteSwap(0x12345678, 0, 3) == byteSwap_standard(0x12345678, 0, 3), "byteSwap(0, 3) failed\n");

    assert(bitCount(0x12345678) == bitCount_standard(0x12345678), "bitCount(0x12345678) failed\n");
    assert(bang(0x12345678) == bang_standard(0x12345678), "bang(0x12345678) failed\n");
    assert(bitParity(0x12345678) == bitParity_standard(0x12345678), "bang(0x12345678) failed\n");
}
```

运行结果:



Microsoft Visual Studio 调试 × + ▾

D:\D2B\ics\exp\lab1\proj\lab1\Debug\task2.exe (进程 26264)已退出, 代码为 0 (0x0)。  
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。  
按任意键关闭此窗口. . .|

没有输出, 说明测试全部通过, 函数功能正确。

## 四、体会

通过本次实验, 我对计算机系统的数据表示与底层运算有了更深刻的理解。在任务 1 中, 通过手动实现结构体的紧凑存储与解压, 我直观体会到内存对齐机制对存储空间的影响。使用逐字节复制与整体写入两种方式时, 发现虽然后者效率更高, 但需要更精准的指针操作技巧。在调试过程中, 通过内存窗口对比十六进制数据, 验证了小端存储模式下 short 和 float 类型的存储规律, 这让我对数据在内存中的物理表示有了具象认知。

任务 2 的位运算设计极具挑战性。在实现 `absVal()` 时, 通过符号位掩码运算替代条件判断, 让我认识到位运算的高效性; `bitCount()` 的分治累加策略则启发我采用并行化思维处理复杂位操作。特别在实现 `bang()` 函数时, 利用 0 与其它数在补码表示上的特性, 仅用位运算

就实现了逻辑非功能，这种巧妙的数学转换让我对计算机的底层逻辑设计叹服不已。

本次实验也暴露出我的不足：在初始设计 `bitMask()` 时，未充分考虑高低位顺序导致掩码生成错误，后通过二进制展开调试才定位问题。这让我意识到严谨的边界条件测试在系统编程中的重要性。这些实践经验将为我后续学习操作系统和编译原理课程奠定坚实的底层基础。

## 五、源码

实验任务 1、2 的源程序（单倍行距，5 号宋体字）

```
#define _CRT_SECURE_NO_WARNINGS
```

```
#pragma pack(1)
```

```
#include <stdio>
```

```
#include <string>
```

```
#include <vector>
```

```
const int N = 5;
```

```
const int N1 = 2;
```

```
const int N2 = 3;
```

```
struct student {
```

```
    char name[8];
```

```
    short age;
```

```
    float score;
```

```
    char remark[200]; // 备注信息
```

```
};
```

```
student old_s[N], new_s[N];
```

```
std::vector<char> msg_vec;
```

```
int pack_student_bytebybyte(student* s, int sno, char*& buf);
```

```
int pack_student_whole(student* s, int sno, char*& buf);
```

```
int restore_student(char* buf, int len, student* s);
```

```
void print_student(student* s, int sno)
```

```
{  
    printf("%-9s %-3s %-5s %-32s\n", "name", "age", "score", "remark");  
    for (int i = 0; i < sno; ++i) {  
        //printf("%-9s %-3hd ", s[i].name, s[i].age, s[i].score, s[i].remark);  
        printf("%-9s %-3hd %-5.1f %-32s\n", s[i].name, s[i].age, s[i].score, s[i].remark);  
    }  
}
```

```
/*
```

```
彭冲 18 96 Excellent
```

```
Jerry 19 88.5 Good
```

```
Rose 20 77.5 Fair
```

```
Jack 21 66.5 Poor
```

```
Lily 22 55.5 Fail
```

```
*/
```

```
int scan_student(student* s, int sno)
```

```
{  
    printf("Please input %d students' information(name, age, score, remark), seperate with space:\n", sno);  
    for (int i = 0; i < sno; ++i) {  
        scanf("%s %hd %f %s", s[i].name, &s[i].age, &s[i].score, s[i].remark);  
        // sscanf("Tom 18 99.5 Excellent", "%s %hd %f %s", s[i].name, &s[i].age, &s[i].score, s[i].remark);  
    }  
}
```

```
    unsigned size = sizeof(student) * sno;

    printf("\nScanned %d students' information\nRaw size: %u bytes.\n\n", sno, size);

    return size;
}

int main()
{
    int raw_size = scan_student(old_s, N);

    char* buf1, * buf2;

    int len1 = pack_student_bytebybyte(old_s, N1, buf1);
    int len2 = pack_student_whole(old_s + N1, N2, buf2);

    printf("\nCompressed size: %u bytes.\nCompress ratio: %.2f%%\n\n", len1 + len2, 100. * (len1 + len2) /
raw_size);

    restore_student(buf1, len1, new_s);
    restore_student(buf2, len2, new_s + N1);

    print_student(new_s, N);

    delete[] buf1;
    delete[] buf2;

    return 0;
}

int pack_student_bytebybyte(student* s, int sno, char*& buf)
{

```

```
std::vector<char> aux;

char* p;

const int size = sizeof(student);

for (int i = 0; i < sno; ++i) {
    p = (char*)(s + i);

    // 姓名
    for (int k = 0; k < 8; ++k) {
        if (p[k] != 0) {
            aux.push_back(p[k]);
        }
        else {
            break;
        }
    }
    aux.push_back(0);
    p += 8;

    // 年龄+分数
    aux.push_back(*p);
    for (int k = 2; k < 6; k++) {
        aux.push_back(p[k]);
    }
    p += 6;

    // 评语
    for (int k = 0; k < 200; ++k) {
```

```
        if (p[k] != 0) {
            aux.push_back(p[k]);
        }
        else {
            break;
        }
    }
    aux.push_back(0);
}
```

```
int len = aux.size();
buf = new char[len];
memcpy(buf, aux.data(), len);
```

```
return len;
```

```
}
```

```
int pack_student_whole(student* s, int sno, char*& buf)
```

```
{
```

```
    int len = 0;
```

```
    for (int i = 0; i < sno; ++i) {
```

```
        len += strlen(s[i].name) + 1;
```

```
        len += 1 + (s[i].age & 0xFF00) ? 2 : 1;
```

```
        len += sizeof(float); // float 没有压缩空间，不用加分隔符
```

```
        len += strlen(s[i].remark) + 1;
```

```
    }
```

```
    buf = new char[len];
```

```
    char* p = buf;
```

```
for (int i = 0; i < sno; ++i) {
    strcpy(p, s[i].name);
    p += strlen(s[i].name) + 1;

    memcpy(p, &s[i].age, 1);
    p++;
    memcpy(p, &s[i].score, sizeof(float));
    p += sizeof(float);

    strcpy(p, s[i].remark);
    p += strlen(s[i].remark) + 1;
}

return len;
}

int restore_student(char* buf, int len, student* s)
{
    char* p = buf;

    int sno = 0;

    while (p < (buf + len)) {
        char* pname = s[sno].name;
        while (*p != '\0') {
            *(pname++) = *(p++);
        }
        if (pname - s[sno].name <= 7) {
            *pname = '\0';
        }
    }
}
```



```
p++;
```

```
char* p_age = (char*)&s[sno].age;
```

```
p_age[0] = *(p++);
```

```
p_age[1] = 0;
```

```
for (int k = 0; k < sizeof(float); ++k) {
```

```
    p_age[k + 2] = *(p++);
```

```
}
```

```
char* p_remark = s[sno].remark;
```

```
while (*p != '\0') {
```

```
    *(p_remark++) = *(p++);
```

```
}
```

```
*p_remark = '\0';
```

```
p++;
```

```
sno++;
```

```
}
```

```
return sno;
```

```
}
```

```
#include <iostream>
```

```
#include <cassert>
```

```
int absVal(int x) {  
    // 若  $x \geq 0$ ,  $x = x \oplus 0x00000000 + 0$ , mask & 1 = 0  
    // 若  $x < 0$ ,  $x = x \oplus 0xFFFFFFFF + 1$ , mask & 1 = 1  
    int mask = x >> 31;  
    return (x ^ mask) + (mask & 1);  
}
```

```
int absVal_standard(int x) { return (x < 0) ? -x : x; }
```

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

```
int negate_standard(int x) { return -x; }
```

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

```
int bitAnd_standard(int x, int y) { return x & y; }
```

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

```
int bitOr_standard(int x, int y) { return x | y; }
```

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

```
int bitXor_standard(int x, int y) { return x ^ y; }
```

```

int isTmax(int x) {
    return !(x ^ 0x7FFFFFFF);
}

int isTmax_standard(int x) { return x == 0x7FFFFFFF; }

int bitCount(int x) {
    // 首先将 x 的相邻位两两相加
    x = (x & 0x55555555) + ((x >> 1) & 0x55555555); // 0101
    // 然后将相邻的 2 位组两两相加
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333); // 0011
    // 将相邻的 4 位组两两相加
    x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F); // 0000 1111
    // 将相邻的 8 位组两两相加
    x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF); // 0000 0000 1111 1111
    // 将相邻的 16 位组两两相加
    x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF);

    return x;
}

int bitCount_standard(int x) {
    int cnt = 0;
    for (int i = 0; i < 32; i++) {
        if (x & 1) cnt++;
        x >>= 1;
    }
    return cnt;
}

int bitMask(int highbit, int lowbit) {

```

```

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

int bitMask_standard(int highbit, int lowbit) {
    int x = 0;

    for (int i = lowbit; i <= highbit; ++i) {
        x |= (1 << i);
    }

    return x;
}

int addOK(int x, int y) {
    int s = (((x + y) ^ x) >> 31) + 1; // x+y 同号 1, 异号 0
    int t = ((x ^ y) >> 31) + 1; // x y 同号:1, 异号: 0
    return t & (~s);
}

int addOK_standard(int x, int y) {
    if ((x < 0 && y < 0 && x + y >= 0) || (x >= 0 && y >= 0 && x + y < 0)) return 1;
    return 0;
}

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

    int n_byte = x & (0xFF << n * 8);
    int m_byte = x & (0xFF << m * 8);
    x ^= (n_byte | m_byte);

    n_byte = n_byte >> n * 8 << m * 8;
    m_byte = m_byte >> m * 8 << n * 8;
    x |= (n_byte | m_byte);
}

```

```
    return x;
}

int byteSwap_standard(int x, int n, int m) {
    char* t = (char*)&x;
    char a = t[3 - n];
    char b = t[3 - m];
    t[3 - n] = b;
    t[3 - m] = a;
    return x;
}
```

```
int bang(int x) {
    int a = (x >> 31) & 1;
    int b = ((~x + 1) >> 31) & 1;
    return (a | b) + ~1 + 1;
}
```

```
int bang_standard(int x) { return !x; }
```

```
int bitParity(int x) {
    x = (x & 0xFFFF) ^ (x >> 16);
    x = (x & 0xFF) ^ (x >> 8);
    x = (x & 0xF) ^ (x >> 4);
    x = (x & 0x3) ^ (x >> 2);
    return (x & 0x1) ^ (x >> 1);
}
```

```
int bitParity_standard(int x) {
    int cnt = 0;
    for (int i = 0; i < 32; i++) {
        cnt ^= (x & 1);
        x >>= 1;
    }
}
```

```
    }

    return cnt;
}

void test() {
    assert(absVal(8) == absVal_standard(8), "absVal(+) failed\n");
    assert(absVal(-8) == absVal_standard(-8), "absVal(-) failed\n");

    assert(negate(8) == negate_standard(8), "negate(+) failed\n");
    assert(negate(-8) == negate_standard(-8), "negate(-) failed\n");

    assert(bitAnd(0x12345678, 0x87654321) == bitAnd_standard(0x12345678, 0x87654321), "bitAnd()
failed\n");

    assert(bitOr(0x12345678, 0x87654321) == bitOr_standard(0x12345678, 0x87654321), "bitOr() failed\n");

    assert(bitXor(0x12345678, 0x87654321) == bitXor_standard(0x12345678, 0x87654321), "bitXor()
failed\n");

    assert(isTmax(0x12345678) == isTmax_standard(0x12345678), "isTmax(0x12345678) failed\n");
    assert(isTmax(0x7FFFFFFF) == isTmax_standard(0x7FFFFFFF), "isTmax(0x7FFFFFFF) failed\n");

    assert(bitMask(5, 3) == bitMask_standard(5, 3), "bitMask(5, 3) failed\n");

    assert(addOK(0x12345678, 0x87654321) == addOK_standard(0x12345678, 0x87654321), "addOK(+,+)
failed\n");

    assert(addOK(0x12345678, 0x7F654321) == addOK_standard(0x12345678, 0x7F654321), "addOK(+,+)
failed\n");

    assert(addOK(0xFFFFFFFF, 0x80000000) == addOK_standard(0xFFFFFFFF, 0x80000000), "addOK(-,-)
```

```
failed\n");
```

```
assert(byteSwap(0x12345678, 0, 3) == byteSwap_standard(0x12345678, 0, 3), "byteSwap(0, 3) failed\n");
```

```
assert(bitCount(0x12345678) == bitCount_standard(0x12345678), "bitCount(0x12345678) failed\n");
```

```
assert(bang(0x12345678) == bang_standard(0x12345678), "bang(0x12345678) failed\n");
```

```
assert(bitParity(0x12345678) == bitParity_standard(0x12345678), "bang(0x12345678) failed\n");
```

```
}
```

```
int main()
```

```
{
```

```
    test();
```

```
    return 0;
```

```
}
```