華中科技大學

# 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：本硕博 202301

学　　号：U202315752

姓　　名：陈宇航

指导教师：李海波


2024 年 9 月 15 日

## 一、实验目的与要求

⑴ 熟练掌握程序开发的基本方法，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。

（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 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;}

(2) int negate(int x); 不使用负号，实现 -x

判断函数： int netgate_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

# 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：

任务一要求我们去压缩和解压缩，简单地来说，压缩就是将不同数据类型的内存为最小内存寻址单元之后，压缩到一个 char 类型的 message 数组里面。这个过程主要是把 name 数组和 remark 数组里面多余的，没有利用完的空间和 short 与 float 之间的两个对齐字节给删除。解压就是倒过来，通过数据类型和内存，从最小单元一个个读出来，下图是我的结构体定义

```
student old_s[N] = {
    {"francis", 19, 43, "mystery"},
    {"love", 1, 100, "disappear"},
    {"someone", 19, 100, "where"},
    {"who", 57, 99, "donot"},
    {"lost", 99, 0, "always"}
};
```

图 3.1.1 结构体内容定义

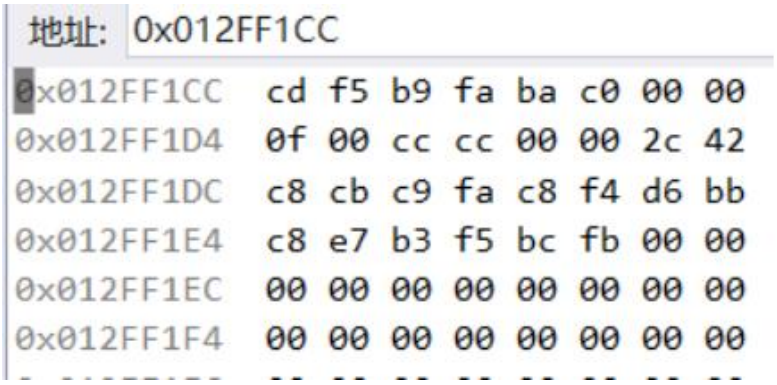


图 3.1.2 结构体内容内存

可以观察到有很多字节段显示为“00”，这些都是没有被使用的内存，也是我们要压缩的空间，下面我们会给出如何压缩

**pack_student_bytebybyte 函数实现：**

该函数的主要作用是对 student 结构体中的数据进行字节压缩，将学生信息逐个字段存储到缓冲区 buf 中，并返回压缩后的字节数。函数参数中，s 是指向 student 数组的指针，sno 是需要压缩的学生数量，buf 是压缩后的存储地址。每次从 student 结构体中读取数据，依次读取名字、年龄、成绩和备注，并将这些数据写入到 buf 中，最后返回总共压缩的字节数 cntbuf。名字最多有 10 个字符。通过循环读取每个字符，直到遇到字符串的结束符（\0），则说明名字结束。此时，将结束符 \0 写入 buf 并结束名字的读取。如果名字的实际长度不足 10 个字符，则跳过剩余的空间（最多跳过 8 - cntname 个字节，以保证字节对齐）。需要注意，因为 short 和 float 类型在内存中对齐到 8 字节边界，这里通过 p += 2 跳过对齐产生的 2 个字节的空余。

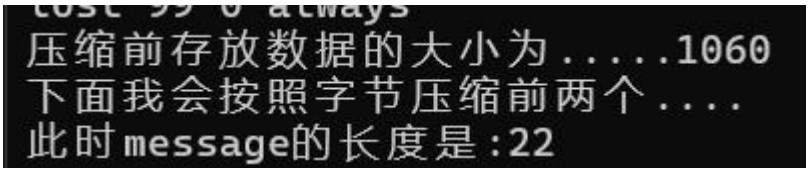


图 3.1.3 结构体压缩结果 1

**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。


图 3.1.4 结构体压缩结果 2

**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：

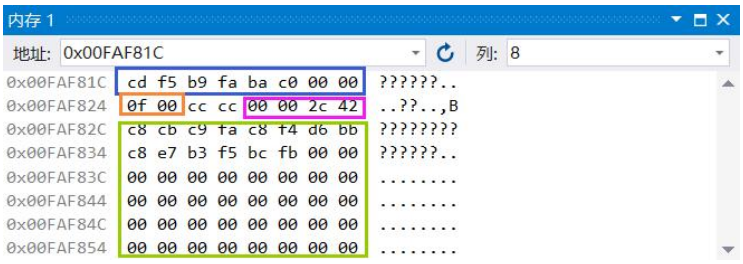


图 3.1.5 结构体数组内存信息

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：

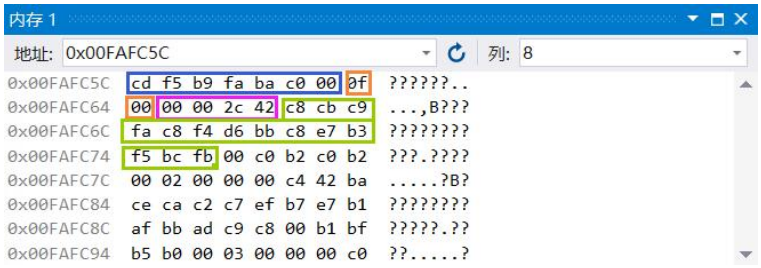


图 3.1.6 message 数组内存信息

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:

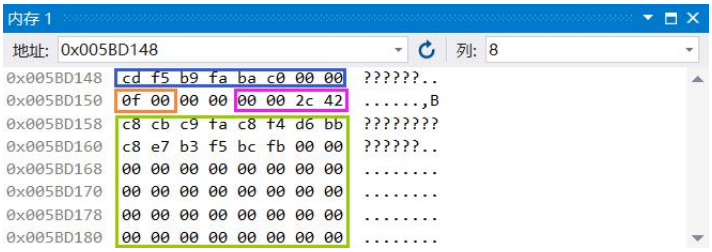


图 3.1.7 message 数组解压内存信息

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：


图 3.1.8 message 前 20 字节内容

下面我们在 vs 的调试窗口查看 message 的前 20 个内容：

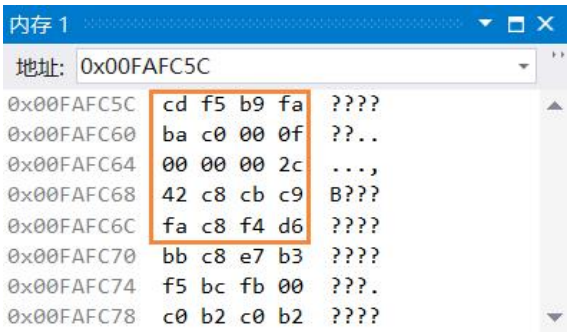


图 3.1.9 message 数组解压内存信息

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 = $1.01011 * 2^5$，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：

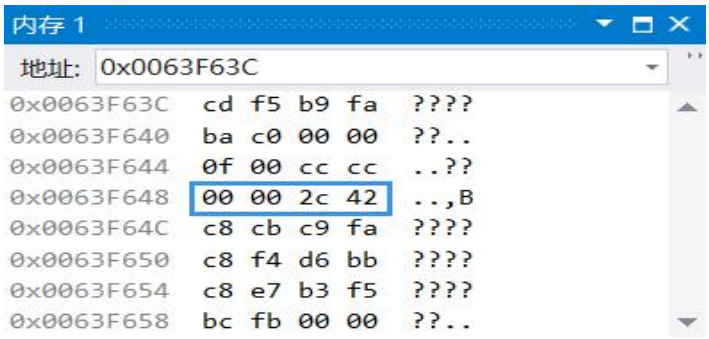


图 3.1.10 浮点编码验证

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

图 3.2.1 absVal 函数实现

```
请输入一个数，我会转化为绝对值输出并匹配....
0
建立的函数返回值是: 0
判断函数返回值是: 0
匹配成功
请按任意键继续. . .
```

图 3.2.2 absVal 函数验证

```
请输入一个数，我会转化为绝对值输出并匹配....
982
建立的函数返回值是: 982
判断函数返回值是: 982
匹配成功
请按任意键继续. . .
```

图 3.2.3 absVal 函数验证

```
请输入一个数，我会转化为绝对值输出并匹配....
-19
建立的函数返回值是: 19
判断函数返回值是: 19
匹配成功
请按任意键继续. . .
```

图 3.2.4 absVal 函数验证

## int netgate(int x)函数实现分析：

这段代码定义了一个名为 netgate 的函数，接受一个整数参数 x，并通过返回表达式 (~x) + 1 实现将 x 转换为其相反数的功能。具体来说，~x 是对 x 进行按位取反操作，将所有二进制位反转，然后加上 1，这实际上是计算机中表示负数的方式，即二进制补码的应用。因此，整个函数的效果等价于返回 -x，即将输入的整数变为其负值。

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

图 3.2.5 netgate 函数实现

```
请输入一个数，我会转化为负数输出并匹配....
-88
建立的函数返回值是: 88
判断函数返回值是: 88
匹配成功
请按任意键继续. . .
```

图 3.2.4 netgate 函数验证

```
请输入一个数，我会转化为负数输出并匹配....
0
建立的函数返回值是：0
判断函数返回值是：0
匹配成功
请按任意键继续. . .
```

图 3.2.5 netgate 函数验证

```
请输入一个数，我会转化为负数输出并匹配....
88
建立的函数返回值是：-88
判断函数返回值是：-88
匹配成功
请按任意键继续. . .
```

图 3.2.6 netgate 函数验证

## int bitAnd(int x,int y)函数实现分析：

这段代码定义了一个名为 bitAnd 的函数,接受两个整数参数 x 和 y,并返回它们的按位与操作结果。其逻辑是通过先对 x 和 y 分别进行按位取反操作，然后对取反后的结果进行按位或操作，最后再对结果进行按位取反。这实际上是利用德摩根定律的变换，间接地实现了 x & y 的功能，因此这个函数的作用是返回 x 和 y 的按位与操作的结果。

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

图 3.2.7 bitAnd 函数实现

```
请输入2个数，我会转化为按位与输出并匹配....
4567 512
建立的函数返回值是：0
判断函数返回值是：0
匹配成功
请按任意键继续. . .
```

图 3.2.8 bitAnd 函数验证

## int bitOr(int x, int y)函数实现分析：

这段代码定义了一个名为 bitOr 的函数，接受两个整数参数 x 和 y，并返回它们的按位或操作结果。其逻辑是通过先对 x 和 y 分别进行按位取反操作，然后对取反后的结果进行按位与操作，最后再对结果进行按位取反。该过程利用了德摩根定律的变换，间接地实现了 x | y 的功能，因此这个函数的作用是返回 x 和 y 的按位或操作的结果。

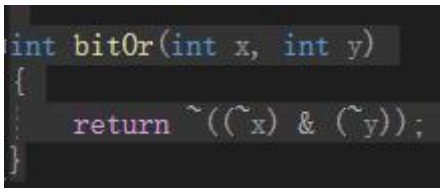


图 3.2.9 bitOr 函数实现

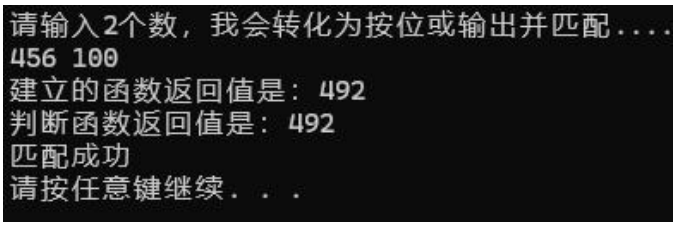


图 3.2.10 bitOr 函数验证

## int bitXor(int x, int y)函数实现分析：

这段代码定义了一个名为 bitXor 的函数，接受两个整数参数 x 和 y，并返回它们的按位异或操作结果。其逻辑是通过使用按位取反和按位与的组合来间接实现 x ^ y 的功能。首先对 x 进行取反并与 y 进行按位与操作，再对 x 与取反后的 y 进行按位与操作，接着对这两个结果进行按位与操作，最后将整个结果取反。这个复杂的变换实际上是通过德摩根定律和基本的逻辑运算来模拟 x ^ y 的行为，因此函数的作用是返回 x 和 y 的按位异或操作结果。

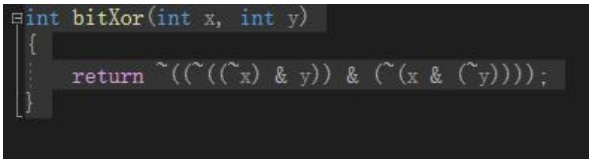


图 3.2.11 bitXor 函数实现

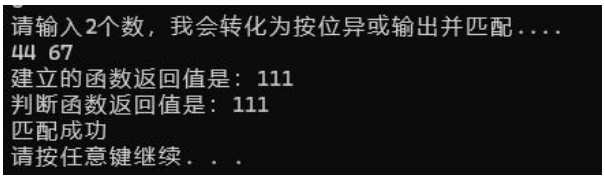


图 3.2.12 bitXor 函数验证

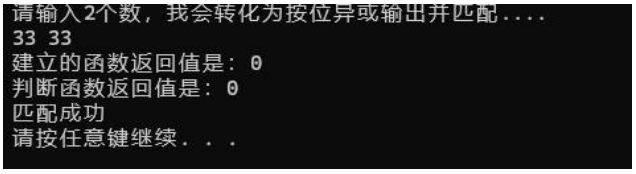


图 3.2.13 bitXor 函数验证

## int isTmax(int x)函数实现分析：

这段代码定义了一个函数 `isTmax`，用来判断给定的整数 `x` 是否为有符号整数类型的最大值 `Tmax`（通常为 `2^31 - 1` 或 `0x7FFFFFFF`）。函数的逻辑是通过以下步骤实现的：

1. `1 << 31`：这会将数字 1 左移 31 位，结果是一个二进制数，最高位为 1，其他位为 0，即 `2^31`（十进制为 2147483648）。

2. `x + (1 << 31)`：将 `x` 加上 `2^31`，如果 `x` 是 `Tmax`，则加上 `2^31` 后结果会是 `0xFFFFFFFF`。

3. `~(x + (1 << 31))`：对这个结果进行按位取反操作。如果 `x + (1 << 31)` 为 `0xFFFFFFFF`（全 1），则取反后的结果为 0。

4. `!(...)`：对取反后的结果应用逻辑非运算。如果结果为 0，逻辑非将返回 1（即 `x` 是 `Tmax`）；否则返回 0。

因此，当 `x` 等于 `Tmax` 时，函数返回 1，表示 `x` 是最大值；如果 `x` 不是 `Tmax`，则返回 0。

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

图 3.2.14 isTmax 函数实现

```
请输入一个数，我会判断是不是最大正数输出并匹配....
2147483647
建立的函数返回值是：1
判断函数返回值是：1
匹配成功
请按任意键继续. . .
```

图 3.2.15 isTmax 函数验证

```
请输入一个数，我会判断是不是最大正数输出并匹配....
52487
建立的函数返回值是：0
判断函数返回值是：0
匹配成功
请按任意键继续. . .
```

图 3.2.16 isTmax 函数验证

## nt bitCount(int x)函数实现分析：

这段代码定义了一个名为 `bitCount` 的函数，用来计算整数 `x` 的二进制表示中有多少个 1（即 Hamming 权重）。该算法通过分层逐步累加位来实现这一点，利用了掩码和位移操作，具体步骤如下：

1. **掩码定义**：
   - `mask1` 用于隔位统计每两个相邻位中 1 的个数，它是 32 位掩码，模式为 `01010101...`。
   - `mask2` 用于每四个位统计 1 的个数，它是 32 位掩码，模式为 `00110011...`。
   - `mask4` 用于每八个位统计 1 的个数，它是 32 位掩码，模式为 `00001111...`。
   - `mask8` 和 `mask16` 则分别用于统计每 16 位和每 32 位的 1 的个数。
2. **位统计过程**：
   - **第一步**：通过 `x = (x & mask1) + ((x >> 1) & mask1)`，将 `x` 每两个相邻位中的 1 的个数相加，并存放到相应的位中。
   - **第二步**：通过 `x = (x & mask2) + ((x >> 2) & mask2)`，将 `x` 中每四个位的 1 的个数相加。
   - **第三步**：通过 `x = (x & mask4) + ((x >> 4) & mask4)`，统计每八个位中 1 的个数。
   - **第四步**：通过 `x = (x & mask8) + ((x >> 8) & mask8)`，统计每 16 位的 1 的个数。

- **第五步**：通过 `x = (x & mask16) + ((x >> 16) & mask16)`，最后将 32 位的 1 的个数汇总到结果中。

最终返回值就是 `x` 中 1 的个数。这种算法称为**分治算法**，通过逐步归并每一位的信息，减少操作次数。

```
int bitCount(int x) {
    // 定义掩码
    int mask1 = 0x55 | (0x55 << 8);  // 01010101 01010101 (16-bit)
    mask1 = mask1 | (mask1 << 16);   // 01010101 01010101 01010101 01010101 (32-bit)

    int mask2 = 0x33 | (0x33 << 8);  // 00110011 00110011 (16-bit)
    mask2 = mask2 | (mask2 << 16);   // 00110011 00110011 00110011 00110011 (32-bit)

    int mask4 = 0x0F | (0x0F << 8);  // 00001111 00001111 (16-bit)
    mask4 = mask4 | (mask4 << 16);   // 00001111 00001111 00001111 00001111 (32-bit)

    int mask8 = 0xFF | (0xFF << 16); // 11111111 00000000 11111111 00000000 (32-bit)
    int mask16 = 0xFF | (0xFF << 8); // 00000000 11111111 00000000 11111111 (32-bit)

    // 第一步：每两个位统计一次1的个数
    x = (x & mask1) + ((x >> 1) & mask1);

    // 第二步：每四个位统计一次1的个数
    x = (x & mask2) + ((x >> 2) & mask2);

    // 第三步：每八个位统计一次1的个数
    x = (x & mask4) + ((x >> 4) & mask4);

    // 第四步：每16个位统计一次1的个数
    x = (x & mask8) + ((x >> 8) & mask8);

    // 第五步：每32位统计一次1的个数
    x = (x & mask16) + ((x >> 16) & mask16);

    return x;
```

图 3.2.16 bitCount 函数实现

```
请输入一个数，我会统计二进制中1的个数并匹配....
44
建立的函数返回值是：3
判断函数返回值是：3
匹配成功
请按任意键继续. . .
```

图 3.2.16 bitCount 函数验证

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

这段代码定义了一个名为 `bitMask` 的函数，接受两个整数参数 `lowbit` 和 `highbit`，返回一个从 `lowbit` 到 `highbit` 之间的位全为 1，其他位为 0 的掩码。具体逻辑如下：

1. **生成从 0 到 highbit 全为 1 的数**：

`int highMask = (1 << (highbit + 1)) - 1;`

通过左移 `1`，将其移到 `highbit + 1` 位，得到一个形如 `10000...000` 的数。减去 1 后，将前 `highbit + 1` 位全变为 1（如 `1111...111`），而其他位为 0。

2. **生成掩码，去掉 lowbit 以下的位**：

`int lowMask = (1 << lowbit) - 1;`

这里通过左移 `1`，将其移到 `lowbit` 位，得到一个形如 `1000...000` 的数，减去 1 后，将 `lowbit` 以下的位全变为 1，而其他位为 0。

3. **生成最终掩码**：

`return highMask & ~lowMask;`

通过对 `lowMask` 进行按位取反（即 `~lowMask`），使得 `lowbit` 以下的位全为 0，而其他位为 1。将其与 `highMask` 进行按位与操作，只保留 `lowbit` 到 `highbit` 之间的位为 1，其他位为 0。

最终，函数返回的结果是一个二进制掩码，位于 `lowbit` 和 `highbit` 之间的位全为 1，其他位为 0。

```
int bitMask(int lowbit, int highbit) {
    // Step 1: 生成从 0 到 highbit 全为 1 的数
    int highMask = (1 << (highbit + 1)) - 1;

    // Step 2: 生成掩码，去掉 lowbit 以下的位
    int lowMask = (1 << lowbit) - 1;

    // Step 3: 将两者相与，保留从 lowbit 到 highbit 的 1
    return highMask & ~lowMask;
}
```

图 3.2.17 bitMask 函数实现

```
我会把从lowerbit到higherbit的值变成1并匹配....
请输入lowerbit和highbit
4 9
建立的函数返回值是: 1008
判断函数返回值是: 0
请按任意键继续. . .
```

图 3.2.18 bitMask 函数验证

```
我会把从lowerbit到higherbit的值变成1并匹配....
请输入lowerbit和highbit
3 6
建立的函数返回值是: 120
判断函数返回值是: 0
请按任意键继续. . .
```

图 3.2.19 bitMask 函数验证

## int addOK(int x, int y)函数实现分析：

这段代码定义了一个名为 `addOK` 的函数，用来判断两个整数 `x` 和 `y` 的加法操作是否会产生溢出。溢出发生在有符号整数加法中，尤其是当两个数符号相同，而结果的符号与它们不同的情况下。具体逻辑如下：

1. **计算 `x + y` 的和**：

`int sum = x + y;`

这一步将两个整数 `x` 和 `y` 相加，得到结果 `sum`。

2. **提取符号位**：

`int x_sign = x >> 31;`

`int y_sign = y >> 31;`

`int sum_sign = sum >> 31;`

通过将 `x`、`y` 和 `sum` 向右移 31 位，得到它们的符号位（第 31 位），符号位为 1 表示负数，为 0 表示非负数。

3. **判断溢出**：

`int overflow = !(x_sign ^ y_sign) & (x_sign ^ sum_sign);`

首先，`x_sign ^ y_sign` 判断 `x` 和 `y` 的符号是否相同，如果相同则返回 0（不会溢出），不同则返回 1（可能溢出）。`x_sign ^ sum_sign` 用来判断 `x` 和 `sum` 的符号是否不同。如果 `x` 和 `y` 的符号相同，但 `sum` 的符号不同，说明发生了溢出。因此，`!(x_sign ^ y_sign) & (x_sign ^ sum_sign)` 表示只有在 `x` 和 `y` 符号相同，并且 `sum` 的符号与它们不同的情况下，溢出才发生。

4. **返回结果**：

`return overflow;`

如果溢出发生，`overflow` 为 1，函数返回 1 表示溢出；如果没有溢出，`overflow` 为 0，函数返回 0。总结来说，`addOK` 函数判断加法是否安全，如果溢出则返回 1，否则返回 0。

```
int addOK(int x, int y) {
    int sum = x + y;

    // 获取x和y的符号位（符号位是第31位）
    int x_sign = x >> 31;
    int y_sign = y >> 31;

    // 获取sum的符号位
    int sum_sign = sum >> 31;

    // 如果 x 和 y 的符号相同，且 sum 的符号与 x 或 y 的符号不同，则发生溢出
    int overflow = !(x_sign ^ y_sign) & (x_sign ^ sum_sign);

    // 如果发生溢出，返回1，否则返回0
    return overflow;
}
```

图 3.2.19 addOK 函数实现

```
我会判断两个数想加会不会溢出并匹配....
请输入2个数字
666 666
建立的函数返回值是: 0
判断函数返回值是: 0
匹配成功
请按任意键继续. . .
```

图 3.2.20 addOK 函数验证

## 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。

```
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)));
}
```

图 3.2.21 byteSwap 函数实现

```
我会把一个数的第n个字节和第m个字节交换并匹配....
请输入1个数字x, 和n, m
854 3 0
建立的函数返回值是: 14428413444
判断函数返回值是: 14428413444
匹配成功
```

图 3.2.22 byteSwap 函数验证

# 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

## 任务一源码

```
#include<bits/stdc++.h>
#include<windows.h>
using namespace std;


const int N = 5;


struct student {
    char name[8];
    short age;
    float score;//
    char remark[196];//备注信息
};


student new_s[N];

//两种压缩函数 返回压缩后的长度
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);




int main()
{
    char message[500];
```

```
    student old_s[N] = {
        {"francis",19,43,"mystery"},
        {"love",1,100,"disappear"},
        {"someone",19,100,"where"},
        {"who",57,99,"donot"},
        {"lost",99,0,"always"}
    };
    cout << "现在我们会输出没有压缩的内容" << endl;
    //开始打印 old_s;
    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 << "压缩前存放数据的大小为....." << sizeof(old_s) << endl;
    cout << "下面我会按照字节压缩前两个...." << endl;



    //按照字节进行压缩

    int len = pack_student_bytebybyte(old_s, 2, message);


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

    //一条一条压缩
    cout << "下面我会按照字节压缩后三个...." << endl;
    len += pack_student_whole(&old_s[2], 3, message + len);

    cout << "此时 message 的长度是:." << len << endl;


    //开始输出存储信息后的 message 的信息
    cout << "开始输出存储信息后的 message 的信息....." << endl;
    printMessage(message, len);

    //下面解压 message 中的信息到结构体里面
    cout << "下面解压 message 中的信息到结构体里面..." << endl;
    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 << "下面以 16 进制的形势，输出 message 里面的前 20 个字节的信息" << endl;
    for (int i = 0; i < 20; i++)
    {
        printf("%02x ", (unsigned char)(*(pp + i)));
    }
    return 0;

}
```

```
int pack_student_bytebybyte(student* s, int sno, char* buf)
{
    //s 是要压缩的起始地址 sno 是压缩人数
    //buf 是压缩之后的首地址 返回压缩之后的字节数
    int cnts = 0;
    int cntname, cntage, cntscore, cntmark;
    int cntbuf = 0;//要返回的数字
    char*p = (char*)s;
    char* pp = (char*)buf;
    while (cnts < sno)
    {
        //读取名字
        cntname = 0;
        while (cntname < 10)
        {
            if (*p)
            {
                *pp = *p;
                cntname++;
                cntbuf++;
                p++;
                pp++;
            }
            else
            {
                *pp = '\0';
                cntbuf++;
```

```
            p += (8 - cntname);//跳过 name 数组剩下的没有使用的空间
            pp++;
            break;
        }
    }

    //读取年龄
    cntage = 0;
    while (cntage < 2)
    {
        *pp = *p;
        cntbuf++;
        cntage++;
        p++;
        pp++;
    }

    //内存对齐：short2 个 float4 个 每一段是 8 个 这里会有 2 个对齐
    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
    cntmark = 0;
    while (cntmark < 20)
    {
        if (*p)
        {
            *pp = *p;
            cntbuf++, cntmark++, p++, pp++;
        }
        else
        {
            *pp = 0;
            cntbuf++;
            pp++;
```

```
                p += (200 - cntmark);
                break;
            }
            cnts++;
        }
    }
    return cntbuf;
}


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)
    {
        strcpy(pp, ppp[cnts].name);
        pp += strlen(ppp[cnts].name) + 1;//小心 0
        *((short*)pp) = ppp[cnts].age;
        pp += 2;//补位
        *((float*)pp) = ppp[cnts].score;
        pp += 4;
        strcpy(pp, ppp[cnts].remark);
        pp += strlen(ppp[cnts].remark) + 1;
        cnts++;
    }
    return pp - buf;
}



//buf 是存储的首地址，len 为长度,s 为存放压缩数据的结构数组的手地址，返回解压人数
int restore_student(char* buf, int len, student* s)
{
    int cnt=0;//记录已经解压的人数
    char* p = buf;
    student* pp = s;
    while ((p - buf) < len)
    {
        //解压名字
        strcpy(pp[cnt].name, p);
        p += strlen(pp[cnt].name) + 1;
        pp[cnt].age = *((short*)p);
```

```
        p += 2;
        pp[cnt].score = *((float*)p);
        p += 4;
        strcpy(pp[cnt].remark, p);
        p += strlen(pp[cnt].remark) + 1;
        cnt++;
    }
    return cnt;
}



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) << ' ';
        p += 2;
        cout << *((float*)p) << ' ';
        p += 4;
        cout << p << ' '<<endl;
        p += strlen(p) +1;
    }
}
```

## 任务二源码

```
#include<bits/stdc++.h>

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.不适用负号，返回-x
int netgate(int x);
//评定函数 2
int netgate_standard(int x) { return -x; }
```

```
//3.(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.(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);
//判断函数 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);
//判断函数 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 其他全为 0 的数
int bitMask(int highbit, int lowbit);
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 会产生溢出的时候返回 1，否则返回 0
int addOK(int x, int y);
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。 n、m 的取值在 0~3 之间。
int byteSwap(int x, int n, int m);
//评定函数 10
int byteSwap_standard(int x, int n, int m) {
    // 计算字节的偏移量
    int n_shift = n * 8;
    int m_shift = m * 8;

    // 提取第 n 个字节和第 m 个字节
    int n_byte = (x >> n_shift) & 0xFF;
    int m_byte = (x >> m_shift) & 0xFF;

    // 将第 n 和第 m 个字节位置清零
    x = x & ~(0xFF << n_shift);  // 清零第 n 个字节
    x = x & ~(0xFF << m_shift);  // 清零第 m 个字节

    // 交换字节
    x = x | (n_byte << m_shift);  // 将第 n 个字节放到第 m 位置
    x = x | (m_byte << n_shift);  // 将第 m 个字节放到第 n 位置

    return x;
}


int main()
{
    int n, m, u;
    int x;
```

```
    int num, num1, num2;
    printf("1.返回 x 的绝对值\n"
        "2.不适用负号，返回 - x\n"
        "3.仅使用 ~ 和 |，实现 &\n"
        "4.仅使用 ~和 & ，实现|\n"
        "5.仅使用 ~ 和 &，实现 ^\n"
        "6.判断 x 是否为最大的正整数（7FFFFFFF)\n"
        "7.统计 x 的二进制表示中 1 的个数\n"
        "8.产生从 lowbit 到 highbit 全为 1 其他全为 0 的数\n"
        "9.当 x+y 会产生溢出的时候返回 1，否则返回 0\n"
        "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。 n、m 的取值在 0~3 之间。\n"
        "0.输入 0 退出");
    while (1)
    {
        system("cls");
        printf("1.返回 x 的绝对值\n"
            "2.不适用负号，返回 - x\n"
            "3.仅使用 ~ 和 |，实现 &\n"
            "4.仅使用 ~和 & ，实现|\n"
            "5.仅使用 ~ 和 &，实现 ^\n"
            "6.判断 x 是否为最大的正整数（7FFFFFFF)\n"
            "7.统计 x 的二进制表示中 1 的个数\n"
            "8.产生从 lowbit 到 highbit 全为 1 其他全为 0 的数\n"
            "9.当 x+y 会产生溢出的时候返回 1，否则返回 0\n"
            "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。 n、m 的取值在 0~3 之间。
\n"
            "0.输入 0 退出\n");
        cin >> x;
        switch (x) {
        case 0:
            cout << "正在退出程序" << endl;
            break;
        case 1:
            cout << "请输入一个数，我会转化为绝对值输出并匹配...." << endl;
            cin >> num;
            cout << "建立的函数返回值是：" << absVal(num) << endl;
            cout << "判断函数返回值是：" << absVal_standard(num) << endl;
            if (absVal(num) == absVal_standard(num))
            {
                cout << "匹配成功" << endl;
            }
            break;
        case 2:
            cout << "请输入一个数，我会转化为负数输出并匹配...." << endl;
            cin >> num;
```

```
        cout << "建立的函数返回值是：" << netgate(num) << endl;
        cout << "判断函数返回值是：" << netgate_standard(num) << endl;
        if (netgate(num) == netgate_standard(num))
        {
            cout << "匹配成功" << endl;
        }
        break;
    case 3:
        cout << "请输入 2 个数，我会转化为按位与输出并匹配...." << endl;
        cin >> num1 >> num2;
        cout << "建立的函数返回值是：" << bitAnd(num1, num2) << endl;
        cout << "判断函数返回值是：" << bitAnd_standard(num1, num2) << endl;
        if (bitAnd(num1, num2) == bitAnd_standard(num1, num2))
        {
            cout << "匹配成功" << endl;
        }
        break;
    case 4:
        cout << "请输入 2 个数，我会转化为按位或输出并匹配...." << endl;
        cin >> num1 >> num2;
        cout << "建立的函数返回值是：" << bitOr(num1, num2) << endl;
        cout << "判断函数返回值是：" << bitOr_standard(num1, num2) << endl;
        if (bitOr(num1, num2) == bitOr_standard(num1, num2))
        {
            cout << "匹配成功" << endl;
        }
        break;
    case 5:
        cout << "请输入 2 个数，我会转化为按位异或输出并匹配...." << endl;
        cin >> num1 >> num2;
        cout << "建立的函数返回值是：" << bitXor(num1, num2) << endl;
        cout << "判断函数返回值是：" << bitXor_standard(num1, num2) << endl;
        if (bitXor(num1, num2) == bitXor_standard(num1, num2))
        {
            cout << "匹配成功" << endl;
        }
        break;
    case 6:
        cout << "请输入一个数，我会判断是不是最大正数输出并匹配...." << endl;
        cin >> num;
        cout << "建立的函数返回值是：" << isTmax(num) << endl;
        cout << "判断函数返回值是：" << isTmax_standard(num) << endl;
        if (isTmax(num) == isTmax_standard(num))
        {
            cout << "匹配成功" << endl;
```

```
        }
        break;
    case 7:
        cout << "请输入一个数，我会统计二进制中 1 的个数并匹配...." << endl;
        int num;
        cin >> num;
        cout << "建立的函数返回值是：" << bitCount(num) << endl;
        cout << "判断函数返回值是：" << bitCount_standard(num) << endl;
        if (bitCount(num) == bitCount_standard(num))
        {
            cout << "匹配成功" << endl;
        }
        break;
    case 8:
        cout << "我会把从 lowerbit 到 higherbit 的值变成 1 并匹配...." << endl;
        cout << "请输入 lowerbit 和 highbit\n";
        int l, h;
        cin >> l >> h;
        cout << "建立的函数返回值是：" << bitMask(l, h) << endl;
        cout << "判断函数返回值是：" << bitMask_standard(l, h) << endl;
        if (bitMask(l, h) == bitMask_standard(l, h))
        {
            cout << "匹配成功" << endl;
        }
        break;
    case 9:
        cout << "我会判断两个数想加会不会溢出并匹配...." << endl;
        cout << "请输入 2 个数字\n";
        int num1, num2;
        cin >> num1 >> num2;
        cout << "建立的函数返回值是：" << addOK(num1, num2) << endl;
        cout << "判断函数返回值是：" << addOK_standard(num1, num2) << endl;
        if (addOK(num1, num2) == addOK_standard(num1, num2)) {
            cout << "匹配成功" << endl;
        }
        break;
    case 10:
        cout << "我会把一个数的第 n 个字节和第 m 个字节交换并匹配...." << endl;
        cout << "请输入 1 个数字 x，和 n，m\n";
        int x, n, m;
        cin >> x >> n >> m;
        cout << "建立的函数返回值是：" << byteSwap(x, n, m) << endl;
        cout << "判断函数返回值是：" << byteSwap_standard(x, n, m) << endl;
        if (byteSwap(x, n, m) == byteSwap_standard(x, n, m)) {
            cout << "匹配成功" << endl;
```

```
            }
            break;
        }
        if (x == 0)
        {
            break;
        }
        else
        {
            system("pause");
        }
    }
    return 0;
}

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

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

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


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

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


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

```
}


int bitCount(int x) {
    // 定义掩码
    int mask1 = 0x55 | (0x55 << 8);  // 01010101 01010101 (16-bit)
    mask1 = mask1 | (mask1 << 16);   // 01010101 01010101 01010101 01010101 (32-bit)

    int mask2 = 0x33 | (0x33 << 8);  // 00110011 00110011 (16-bit)
    mask2 = mask2 | (mask2 << 16);   // 00110011 00110011 00110011 00110011 (32-bit)

    int mask4 = 0x0F | (0x0F << 8);  // 00001111 00001111 (16-bit)
    mask4 = mask4 | (mask4 << 16);   // 00001111 00001111 00001111 00001111 (32-bit)

    int mask8 = 0xFF | (0xFF << 16); // 11111111 00000000 11111111 00000000 (32-bit)
    int mask16 = 0xFF | (0xFF << 8); // 00000000 11111111 00000000 11111111 (32-bit)

    // 第一步：每两个位统计一次 1 的个数
    x = (x & mask1) + ((x >> 1) & mask1);

    // 第二步：每四个位统计一次 1 的个数
    x = (x & mask2) + ((x >> 2) & mask2);

    // 第三步：每八个位统计一次 1 的个数
    x = (x & mask4) + ((x >> 4) & mask4);

    // 第四步：每 16 个位统计一次 1 的个数
    x = (x & mask8) + ((x >> 8) & mask8);

    // 第五步：每 32 位统计一次 1 的个数
    x = (x & mask16) + ((x >> 16) & mask16);

    return x;
}


int bitMask(int lowbit, int highbit) {
    // Step 1: 生成从 0 到 highbit 全为 1 的数
    int highMask = (1 << (highbit + 1)) - 1;

    // Step 2: 生成掩码，去掉 lowbit 以下的位
    int lowMask = (1 << lowbit) - 1;

    // Step 3: 将两者相与，保留从 lowbit 到 highbit 的 1
    return highMask & ~lowMask;
```

```
}


int addOK(int x, int y) {
    int sum = x + y;

    // 获取 x 和 y 的符号位（符号位是第 31 位）
    int x_sign = x >> 31;
    int y_sign = y >> 31;

    // 获取 sum 的符号位
    int sum_sign = sum >> 31;

    // 如果 x 和 y 的符号相同，且 sum 的符号与 x 或 y 的符号不同，则发生溢出
    int overflow = !(x_sign ^ y_sign) & (x_sign ^ sum_sign);

    // 如果发生溢出，返回 1，否则返回 0
    return overflow;
}


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)));
                                                }
```