作业一

本次作业一共有3道大题：

- 第1题和第2题是整数与浮点数计算，考察对知识点的掌握；
- 第3题是实践题，通过与实际编程结合，考察对知识点的理解与应用。

Q1. 整数计算

Q1.1. 二进制表示

在所有由 4个`1` 和 4个`0`组成的 8位二进制整数（补码）中，最小的带符号数和最大的带符号数分别是多少？答案请以十进制表示。

带符号数的最高位（二进制位"从右到左"对应于"从低到高"）为符号位，`0`表示正数，`1`表示负数。因此最小的带符号数其符号位为`1`；剩下的`0`和`1`按照贪心原则去排列。

最小的带符号数：`10000111b = -121`
最大的带符号数：`01111000b = 120`

Q1.2. 补码计算

X、Y的数据位宽均为16位。已知$[X]_{\text{补码}}=0\text{x}0033$，$[Y]_{\text{补码}}=0\text{xDE5A}$，则$[X-Y]_{\text{补码}}$和$[X+Y]_{\text{补码}}$的值，计算结果用十六进制补码表示。

> **快速完成十六进制数取反**
>
> 给定一个四位二进制数X，其取反结果为$\bar{X}$，恒有$X+\bar{X}=0\text{xF}$
> 方便记忆的值：$0\text{x}0 \xleftrightarrow{\text{取反}} 0\text{xF}$、$0\text{x}5 \xleftrightarrow{\text{取反}} 0\text{xA}$、$0\text{x}7 \xleftrightarrow{\text{取反}} 0\text{x}8$

于是有 $[-Y]_{\text{补码}} \xrightarrow{\text{取反+1}} 0\text{x}21\text{A}5+0\text{x}1=0\text{x}21\text{A}6$

按补码加法直接计算可得：

$$[X-Y]_{\text{补码}}=0\text{x}21\text{D}9$$
$$[X+Y]_{\text{补码}}=0\text{xDE}8\text{D}$$

Q2. 浮点数计算

Q2.1. 定点数与浮点数哪个多?

给定相同的字长（例如32位），能表示的定点数和浮点数哪个更多？请给出你的理由。

定点数中，不同的 01 串所表示所对应的数完全不同。
浮点数中：

1. 规格化数不同的 01 二进制表示所对应的数不同。
2. 非规格化数不同的 01 二进制表示所对应的数不同，正负 0 除外（两个不同的 01 串表示同一个数）。
3. 存在两个 01 串分别表示正负无穷。
4. 存在多个不同的 01 串表示 `NaN`。

综上，相同字长能表示的定点数较多。

Q2.2. 9位浮点数表示

假设存在一种符合IEEE浮点数标准的9位浮点数，由1个符号位、4位阶码、4位尾数组成，数值表示仍遵循$V = (-1)^s \cdot M \cdot 2^E$。请在下表中填空：

描述	9位二进制表示	M（十进制表示）	E（十进制表示）	V（十进制表示）
3.5				3.5
大于0的最小浮点数				

$\text{Bias} = 2^{e-1} - 1 = 2^{4-1} - 1 = 7$

- 3.5（规格化数）：
  $3.5_{10} = 11.1_2 = 1.11_2 \times 2^1$，于是$E = 1$，$M = 1.11_2 = 1.75_{10}$
- 大于0的最小浮点数（非规格化数）：
  其二进制表示为 0 0000 0001，那么$M = 0.0001_2 = 2^{-4} = 1/16$，$E = 1 - \text{Bias} = -6$，值等于$2^{-4} \times 2^{-6} = 2^{-10} = 1/1024$

描述	9位二进制表示	M（十进制表示）	E（十进制表示）	V（十进制表示）
3.5	0 1000 1100	1.75	1	3.5
大于0的最小浮点数	0 0000 0001	1/16	-6	1/1024

Q3. 实践题

Q3.1. 加法溢出检查

带符号数 `si1` 和 `si2` 符号位相同，先检查二者相加后是否会产生加法溢出：若溢出返回 `true`，未溢出返回 `false`，并将结果写进 `*sum` 中。

你需要填写 `__表达式(1)__` 和 `__表达式(2)__`（其中 `__表达式(1)__` 要求使用位运算实现，不能调标准库中 `INT_MIN` 的实现），并解释它的工作原理。

```
bool checkAddOF(int si1, int si2, int* sum) {
    unsigned usum = (unsigned)(si1) + si2;
    const unsigned MY_INT_MIN = __表达式(1)__; // 用位运算实现
    if ((__表达式(2)__) & MY_INT_MIN)
        return true;
    else {
        *sum = si1 + si2;
        return false;
    }
}
```

要求通过位运算来为实现 `INT_MIN`，并赋值给 `MY_INT_MIN`。而 `INT_MIN` 的特征为除符号位为 `1`，其余位为 `0`，一般实现方式如下：

```
/* 表达式(1) */
// (1) 仅限于32位整数
const unsigned MY_INT_MIN = 1u << 31;
// (2) 通用实现，但涉及一次乘法运算（O2能优化掉）
const unsigned MY_INT_MIN = 1u << (sizeof(unsigned) * 8 - 1);
```

加法溢出发生于两个同符号整数相加后，得到的结果与被加数异号，如**负数加负数得到正数**。最直接的做法是判断结果的符号位与被加数 `si1` 或 `si2`（因为它们同号）的符号位是否相同。对此我们可以使用异或运算`来实现：

> **异或运算**
>
> 对于两个位 `x` 和 `y`，如果它们同为 `0` 或同为 `1`，则 `x ^ y = 0`（相互抵消了）；如果不相同，则 `x ^ y = 1`。

X	Y	$X \oplus Y$（即X异或Y）
0	0	0
0	1	1
1	0	0
1	1	0

在C语言中，`x ^ y` 表示对`x`和`y`的每一位进行异或运算，自然包括了作为最高位的符号位：

```
/* 表达式(2) */
// (1) 参考解
unsigned result = usum ^ si1;
// (2) 另一种解答，保险但没必要
unsigned result = (usum ^ si1) & (usum ^ si2);
// (3) 如果被加数是正数，直接看usum符号位；反之是负数，usum是正数，就将其符号位取反
unsigned result = (si1 > 0) ? usum : ~usum; // -usum也行
```

可是我们最终只想看符号位的值：假设加法实际没有溢出，但是除符号位外的位非全`0`（比如`0x01234567`），在C语言的`if ()`中，所有非零值对应逻辑值为`true`，则该溢出判断成立——出bug了！

还记得一开始准备的`MY_INT_MIN`吗？它只有符号位为`1`，其余位为`0`："与"上`MY_INT_MIN`，保留符号位的同时置其余位为`0`，这时的结果是否非零完全取决于符号位，符合我们的预期。

```
if (result & MY_INT_MIN) ... // 符号位异号，发生溢出
```

Q3.2. 字节序

在网络传输数据时，发送方需要将本地的数据通过转换为网络字节序后再发送；接收方接收后，需要转换为本地字节序后才能使用。
这个转换可以调用`htonl()`和`ntohl()`由网络序来实现。

> **`htonl()`和`ntohl()`**
>
> `hton`表示由 本地 (**h**ost) 字节序转为 网络 (**n**etwork) 字节序；后缀`l`表示`long`，即32位整数（对应的`s`表示`short`，即16位整数）。

```c
// byte_order.c
#include <arpa/inet.h>
#include <stdio.h>

void printMemory(void *p, int size) {
    unsigned char *p1 = (unsigned char *)p;
    for (int i = 0; i < size; i++) {
        printf("%02x ", p1[i]);
    }
    printf("\n");
}

int main(int argc, char const *argv[]) {
    int a = 0x12345678;
    printf("Original: "); printMemory(&a, sizeof(a));

    a = htonl(a); // <1>
    printf("Try htonl(): "); printMemory(&a, sizeof(a));

    a = ntohl(a); // <2>
    printf("Then ntohl(): "); printMemory(&a, sizeof(a));
    return 0;
}
```

上述示例代码的编译与执行结果如下所示：

```
$ gcc byte_order.c -o byte_order && ./byte_order
Original: 78 56 34 12
Try htonl(): 12 34 56 78
Then ntohl(): 78 56 34 12
```

> 请确保你已按照实验导引在本地配置好Linux环境并测试过 `gcc` 。

Q3.2.1. 本地序 vs 网络序

根据上述执行结果，请问本地的字节序属于哪一种字节序？网络字节序又属于哪一种？

观察 `printMemory()` 的输出：当地址从低向高递增时，如果是低字节对低位，那么就是小端序 (little endian)；如果是低字节对高位，那么就是大端序 (big endian)。

- 本地：小端序
- 网络：大端序

Q3.2.2. 大小端序转换

试补齐下述宏定义中缺失的数值，使得 `htonl()` 能够正确工作，并解释该宏定义的功能。

```
// glibc/bits/byteswap.h

/* Swap bytes in 32-bit value. */
#define __bswap_constant_32(x)
    ((((x) & __空格(1)__) >> 24) | (((x) & __空格(2)__) >> __空格(3)__) | \
     (((x) & __空格(4)__) <<  8) | (((x) & 0x000000ffu) << __空格(5)__))
```

补充：在C语言中，`0x000000ffu`的意思是将`0x000000ff`转换为无符号整数。注意观察，位运算的格式为`((x) & __无符号十六进制数__) >> __十进制常值__`，按照这个格式补齐`__空格(n)__`的值。

> 出于编程习惯，涉及位运算的操作都倾向于按无符号数处理；实际上不添加后缀 `u` 也能正常工作。

`htonl`和`ntohl`的核心在于将小（大）端序转换为大（小）端序，主要实现在`__bswap_constant_32`宏中。

从`(x) & 0x000000ffu`可知取出最低8位，然后移到最高8位，也就是左移$32-8=24$位，因此`__空格(5)__`的值为`24`

此时分析`__空格(1)__`，它应将最高8位移到最低8位上，这个可以通过`& 0xff000000u`完成，因此`__空格(1)__`为`0xff000000u`。

同理可以推断出`__空格(3)__`是将次高8位移到次低8位，`__空格(3)__`填`8`，而`__空格(2)__`为`0x00ff0000u`，`__空格(4)__`可知为`0x0000ff00u`

最终实现如下：

```
// glibc/bits/byteswap.h

/* Swap bytes in 32-bit value. */
#define __bswap_constant_32(x)
    ((((x) & 0xff000000u) >> 24) | (((x) & 0x00ff0000u) >> 8) | \
     (((x) & 0x0000ff00u) <<  8) | (((x) & 0x000000ffu) << 24))
```

Q3.2.3. 思考题

如果调换`a = htonl(a);`（注释`<1>`处）和`ntohl(a)`（注释`<2>`处），程序的输出结果会有变化吗？请解释原因。

由于底层二进制数据的存储方式是一样的，因此在字节序翻转上并无本质区别。查看glibc实现可知其实二者通过`weak_alias`宏关联在一起：

```
// glibc/inet/htonl.c

uint32_t
htonl (uint32_t x)
{
#if BYTE_ORDER == BIG_ENDIAN
  return x;
#elif BYTE_ORDER == LITTLE_ENDIAN
  return __bswap_32 (x);
#else
# error "What kind of system is this?"
#endif
}
weak_alias (htonl, ntohl)
```

什么是 `weak_alias` 宏

其目的是为函数添加一个"弱"别名，与"强"符号函数名区分。

```
//macro weak_alias
#define weak_alias(name, aliasname) _mweak_alias(name, aliasname)
#define _weak_alias(name, aliasname) \
extern __typeof(name) aliasname __attribute__((weak, alias(#name)));
//macro weak_alias end
```

> 在后续课程中会着重讲解，有兴趣的同学可以先阅读CSAPP第7.6.1节。

具体而言，如果调用"被`weak_alias`"的函数（上例中为`ntohl`）时，无同名的"强"符号函数实现，则会调用该别名对应的函数（即`htonl`）。