(a) 目前使用的格里高利历闰年的规则如下：

1. 公元年分非4的倍数，为平年。

2. 公元年分为4的倍数但非100的倍数，为闰年。

3. 公元年分为100的倍数但非400的倍数，为平年。

4. 公元年分为400的倍数为闰年。

请用一个表达式判断某一年是否为闰年。

(b) 输入某一天的年月日，输出下一天的年月日。

(c) 输入某两天的年月日，输出这两天的相距多少天(不考虑公元前，且第一个日期比第二个日期要早)。

(d) 已知1970年1月1日是星期四，输入之后的某一天的年月日，判断它是星期几?

(e) 输入1970年之后任意一年的年份，输出该年的年历。对话如下：(拓展题，不要求每个同学都做)

1. 判断闰年

```
// 返回true(非0)是闰年 false(0)是平年
int is_leap_year(int  year){
    return ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0) );
}

int get_month_days(int year, int month){

}
```

2.
计算下一天是哪一天

```
void next_day(int year, int month, int day){

}
```

思考:
day++是一定要做的
做完后要考虑溢出的情况,如果day已经超出了本月的天数,那么:
1.day设置为1
2.month++

在月份++的情况下,考虑它的溢出情况,也就是说如果month等于13的情况下:
1.month设置为1
2.year++

上面是求解问题的主线,但是这条主线有一条非常重要的分支:
如何计算出本月的天数? 本month月的天数:
1.首先要考虑平年闰年,如果是闰年,2月就是29天
2.推荐的方式是 用一个数组来存储每一个月份的天数
[-1, 31, 28, 31, 30.....] 13个元素

梳理一下思路:
1.day++
2.获取当前月份的天数
3.如果day是最后一天处理溢出,month++
4.如果month++后等于13,那就处理月份的溢出
5.打印第二天的日期

输入某两天的年月日，输出这两天的相距多少天(不考虑公元前，且第一个日期比第二个日期要早)。
思考:

```
int distance(int year1, int month1, int day1, int year2, int month2, int day2){
    int days = 0;  // 相距多少天
}
```

2020,01,02~2022.03.04
思路是:
1.第一个日期的年还剩下多少天,累加到days中
2.第二个日期的年已经过了多少天,累加到days中
3.计算两个日期年份之间,相距年份的天数,累加到days中
4.如果两个日期的年份是一样的,那就需要减去该年一整年的天数,因为是多加的
2020.01.03~2020.03.03

主线完了就思考支线细节
第一个日期的年还剩下多少天,累加到days中
1.需要知道当前月还剩下多少天,于是还是需要知道本月有多少天
2.累加上剩余月份的天数
3.整个过程都需要考虑平年闰年的影响
第二个日期已经过了多少天:
1.day2可以直接累加
2.前面的月份的天数累加

已知1970年1月1日是星期四，输入之后的某一天的年月日，判断它是星期几?

只需要知道输入的年月日和1970.1.1日过去了多少天
然后拿这个天数取余7,如果余数是:
0表示星期4
1表示星期5
2表示星期6
3表示星期日
4表示星期1
5表示星期2
6表示星期3

```
int get_day_of_week(int year, int month, int day){
   // 获取相距的天数
  return

}
```

希望余数,需要days+4再取余7:
0代表星期日
1代表星期1
2代表星期2
.....

```
}
    printf("第%d次录入的字符是: %c\n", count, ch);
    count++;
  while ((ch = getchar()) != '\n') {        // 只要getchar读取到的数据不是换行符,就继续循环
  // 副作用: 赋值,改变了ch变量的取值
  // 主要作用: 表达式要返回的值,这就是getchar的返回值
  // ch = getchar()
  printf("请输入一个字符串: ");
  int count = 0;        // 输入字符串的字符总数
  int ch;        // 表示getchar()每次读到的数据,因为可能返回EOF,所以类型用int
  // 需求: 利用getchar从键盘录入,计算录入的字符串的总字符数
  // 惯用法,为了确保能够读取完毕数据,可以采取循环读的方式,以某种条件作为结束
  // 所以getchar函数的返回值最好用int来接受,EOF不是字符 不要用char接受
  // getchar函数的返回值是包含EOF的,大多数平台EOF可能就等于-1

  //putchar('\n');
  //putchar(c);
  //char c = getchar();
  //printf("请输入一个字符: ");
  // 它一样会录入空格,不会跳. 如果希望能跳过空格,最好还是用scanf
  // getchar函数,和scanf函数一样将数据从stdin缓冲区中读取出来,遇到换行就刷新缓冲区数据到内存中
  // getchar函数是scanf函数的char类型专供版,它的效率更高,所以更推荐使用
  putchar('\n');*/
  putchar(c);
  /*char c = 'a';
  // putchar函数是printf函数的char类型专供版,它的效率更高,所以更推荐使用
  // putchar函数,和printf函数一样将数据写入到stdout,遇到换行就刷新缓冲区数据到屏幕
```

类型转换:
1.隐式类型转换
2.强制类型转换

隐式类型转换:
1.赋值运算中的隐式类型转换，如果赋值运算符右边的值和左边类型不一致，就会进行隐式类型转换
2.函数调用时参数和函数返回值的类型和声明的类型不一致时，就会进行隐式类型转换
3.不同类型组成的表达式会进行隐式类型转换:

1. **当表达式中仅有int以及int以下等级的类型参与时，表达式的结果一律是int(或者unsigned int)**
2. **表达式的最终结果类型是，表达式中取值范围和精度最大的那个操作数的类型。**
3. **同一转换等级的有符号整数和无符号整数一起参与运算时，有符号整数会转换成对应的无符号整数。**

最后注意:

1. **鉴于无符号整数带来的一系列复杂性和坑爹的陷阱，请大家不要在日常代码中使用无符号数。**
2. 假如你有一天能接触到底层开发，有使用无符号整数的需求，也建议不要混合使用无符号整数和有符号整数。

给类型起别名
语法：
typedef 现有类型的名字 别名;

注意：

a. 别名应该是在原有类型名的基础上，更明确更清晰的表明类型作用的，应该具有好的可读性。"只有起错的名字，没有起错的外号。"

b. 为了区分类型名和别名，建议别名和类型名采用不同的命名风格。比如类型名采用"下划线式命名风格"，那么别名可以采用"驼峰命名风格"。

c. 在C语言标准库中，常用"_t"作为后缀结尾表明此类型名是一个别名。常见的如： `size_t` 、 `int32_t` 等。我们也可以模仿采用这种风格，当然关于命名，一切以公司的要求为最高要求。

好处：

1. 提升代码的可读性。这个很容易理解，不多赘述。原类型名往往是一个通用的称呼，而别名是此场景下的一个精准描述。
2. 提升代码的扩展性。这一点在后续数据结构阶段会体现的很明显，在后续课程我们将展开讲解这部分内容。
3. 提升代码的跨平台性移植性。类型别名的语法最重要的用途就是增强代码的跨平台移植性，下面将详细讲一个作用。

a + (b - a >> 1)
先算b-a
再算(b-a)>>1
最后算
a + ((b-a) >>1)
[a, b]
取区间的中点

p是一个指向int变量的指针，其实就是存int变量的地址
int num = *p++

*p++
*(p++)

(p++)后缀自增自减的作用：
1.主要作用：直接返回p的值
2.副作用：在返回p的值之后，给p自增1

(p++)这个表达式在整个大表达式中，起到的作用就是它的主要作用，就是返回p

所以这个(*p++)表达式，对p解引用，返回解引用的结果。然后p自增1
(*p++)表达式：
主要作用：返回*p
副作用：p++

int num = *p++
主要作用：返回*p
副作用：p++,以及把*p赋值给num

对于一个简单的前缀运算表达式： `--a`
主要作用：返回a变量减1后的结果
副作用：给a变量减1

```
int b = --a;
```
把a减1后的结果赋值给b，a的值减1

```
int a = 1;
int b = ++a;    // a = 2, b = 2

printf("%d\n", --b);    // 打印的结果b = 1

printf("a is %d now.\n", a);    // a = 2
printf("b is %d now.\n", b);    // b = 1
```

表达式 `a++` 同样具有主要作用和副作用：

a. 主要作用是：直接返回变量 `a` 的值，这就是此表达式计算出来的一个值。

b. 副作用是：变量 `a` 自加1

```
// 陷阱，坑爹的C语言语法
int a = 1;
// 在一个表达式中多次改变了一个变量的取值
// 这里就会存在一个赋值顺序的问题
// 按照 C 语言标准，同一个变量在同一个条语句之间只能被修改一次，如果多次修改会引发未定义行为。
//a = a++;

// 这是未定义行为,任何结果都有可能,不要猜,也不用想
int b = a++ + --a;// 1 + 1 = 2?
```

按位运算符中，比较需要留意的是按位异或。它具有以下一些非常优秀的性质：

> **a ^ 0 = a;　任何整数异或0得到的都是它本身**
>
> **a ^ a = 0;　任意整数异或自己得到的都是0**
>
> **a ^ b = b ^ a;　异或运算满足交换律**
>
> **(a ^ b) ^ c = a ^ (b ^ c);　异或运算满足结合律**

所有的位运算符组成的表达式都是没有副作用的,都是不会改变变量取值的

如果希望能够改变取值,那就使用复合赋值运算符

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdbool.h>

/*
* 位运算符
*
*/

// 定义一个函数，判断某个整数是否为奇数。
//bool is_odd(int n) {
//
//}

/*
* 0000 0000
* 0000 0001
* 0000 0010
* 0000 0011
* 0000 0100
* 0000 0101
* ....
* 奇数的最后一位 一定是1
*
* 整数和1做运算
* xxxx xxx0
* 0000 0001
* -->
* 1111 1111
* 0000 0001
* ->
* 0000 0001
*
* 1111 1110
* 0000 0001
* ->
* 0000 0000
*/
bool is_odd(int n) {
}

//给定一个正整数，请定义一个函数判断它是否为2的幂(1, 2, 4, 8, 16, ....)。
//bool is_power_of_2(int n) {
//
//
//
//
//
//
//}

/*
* 0000 0001
* 0000 0010
* 0000 0100
* ...
* 2的整数幂,它的2进制,必然只有一个位是1
*
* 0000 0100 4 & 0000 0011 3 = 0
* 0000 1000 8 & 0000 0111 7 = 0
*
*/
bool is_power_of_2(int n) {
}

// 给定一个不为0的整数，编写函数找出它值为1的最低有效位(称之为Last Set Bit)
/*
输入：n = 24
输出：8
解释：24的二进制表示为 11000，值为 1 的最低有效位为 2^3，所以输出8。

0001 1000
&
0000 0001-->
0000 0010-->
0000 0100-->
0000 1000-->
规律:
让这个整数与1,2,4,8..等2的幂做逻辑与运算,当结果就是整数幂本身,这个幂就是Last Set Bit
*/
//int find_last_set_bit(int n) {
//
//
//
//
//
//}



/*
*
* 补码有一个非常优秀的性质:
* x + (-x) = 0 = 1 0000...000(最高位溢出)
*
* 10: 0000 1010
*-10: 1111 0110
* 对于x来说,-x一定在last set bit位以后它们两一定是一样的
* 而且在之前,它们每一位都不一样
* 于是
* x & (-x) = last set bit
*
*/
int find_last_set_bit(int n) {
}


int main(void) {



}
return 0;
    a = a ^ b;    // b3 = a0, a3 = a2 ^ b2 = a0 ^ b0 ^ a0 = 0 ^ b0 = b0
    b = a ^ b;    // a2 = a0 ^ b0, b2 = a1 ^ b1 = a0 ^ b0 ^ b0 = a0 ^ 0 = a0
    a = a ^ b;    // a1 = a0 ^ b0, b1 = b0
    // 初始值是a0, b0
    // 缺点: 不能交换非整数, 可读性非常差, 看不懂
    // 优点: 不用tmp,不占用额外空间就能交换成功
    // 按位运算进行交换
    */
    b = tmp;
    a = b;
    int tmp = a;
    /*
    // 交换
    int b = 20;
    int a = 10;
    // 给定两个不同的整数 a 和 b，请交换它们两个的值。(这里不要定义函数)
    return n & (-n);
    return x;
    }    // (n & x)!=0时,也就是等于x时,此时x就是last set bit
        x <<= 1;
    while ((n & x) == 0) {
    int x = 1;
    // x代表2的整数幂1,2,4,8...
        本身
        0
        0
        0
        0001 1000
    return (n & (n - 1)) == 0;
        4
        2
        1
    return n == 1;
    }    // n==1 n<1
        n /= 2;
    while (n > 1 && n % 2 == 0) {
    // 只要这个数是大于1的且它能够被2整除,那就除以2
    return n & 1;
    */
    * 如果是偶数,最后一位是0,与1做按位与运算,结果就是0
    * 如果是奇数,最后一位是1,与1做按位与运算,结果就是1
    /*
    return n % 2 != 0;
```

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main(void) {
// 给定一个非空的整数数组num
   // 已知某个元素只出现了一次，其余元素均出现两次，那么请你找出这个只出现一次的元素。
   //int nums[] = { 1, 4, 2, 1, 2 , 3,6,3,6,8,8 };

   //// 思路: 把数组中的所有元素都用^链接起来,结果就是那条单身狗,就是那个唯一的元素
   //int result = 0;
   //for (int i = 0; i < 5; i++) {
   //    result ^= nums[i];
   //}

   // 给定一个非空的整数数组nums
   // 已知有两个元素只出现了一次，其余元素均出现两次，那么请你找出这两个只出现一次的元素。
   int nums[] = { 1, 4, 3, 2 ,1, 2 };
   int result = 0;
   for (int i = 0; i < 6; i++) {
      result ^= nums[i];
   }

   // 找到last set bit
   int lsb = result & (-result);


   // 根据lsb进行分组,只需要逻辑分组,不需要物理分组
   int a = 0, b = 0;        // 两个单独出现的元素
   for (int i = 0; i < 6; i++) {
      if (nums[i] & lsb){
         a ^= nums[i];
      }
      else {
         b ^= nums[i];
      }
   }


   // 设唯一的元素是a和n
   // result = a ^ b
   /*
   * a:4 0000 0100
   * b:3 0000 0011
   * a和b必然是有某一位是不一样的
   * 如何找到这一位呢?
   * a ^ b --> 0000 0111
   * a ^ b的结果的位模式,只要是1的位,就意味着在该为a和b是不一样的
   * 于是我们的任务就是:
   * 随便找一个a ^ b结果是1的位
   * 于是我们就找last_set_bit
   * -->
   * 我们就知道在last_set_bit这一位,a和b取值必然是不一样的
   * 于是就分组:
   * last set bit = 1
   * a: 0000 0100 & 0000 0001 = 0
   * b: 0000 0011 & 0000 0001 = 0000 0001
   *
   * 以上完成分组
   * 再次组内用^链接,结果就分别是两个只出现一次的元素
   */


   return 0;
}
```