程算期末理论题复习

- switch 语句在缺少break 时将 Fall through
- ++优先级高于*
- `true`、`false`、`TRUE`、`FALSE` 是合法的变量名，因为在 C 中它们不是关键字
- `char a = 255`，打印后值为 `-1`，因为 `char` 为有符号类型并采用补码表示，其范围为 `-128 ~ 127`。
- 指针可以加常数，减常数，但指针之间可以相减，但不能相加
  - 两个同一类型的指针变量可以相减，意义是**两个指针指向的内存位置之间相隔多少个元素** 注意是元素不是字节数。
  - 不同类型的指针不允许相减
  -
- 指针只有加减操作，没有乘除操作
- [ ]优先级高于 *
- 指针变量需赋值一个可用的地址之后才可以解引用
- 假设有定义如下：`int array[10];` 则该语句定义了一个数组array。其中array的类型是整型指针 ❌
  - array 是指向int array[0]地址的指针
- For the function declaration `void f(char ** p)`, the definition __ of `var` makes the function call `f(var)` incorrect。

  A.

  ```
  char var[10][10];
  ```

  B.

  ```
  char *var[10];
  ```

  C.

  ```
  void *var = NULL;
  ```

  D.

  ```
  char *v=NULL, **var=&v;
  ```

- &a代表数组地址，类型为：int(*)[ ]

  a代表数组0号元素地址，类型为：int*

指向数组的指针：int (*pName)[ ] = &a;

指向数组0号元素的指针：int* pName = a;

当指针指向数组元素时，可以进行指针移动。

即 指针和数组名在效果上是等价的。**区别在于：指针是变量** 指针可以参与表达式的计算，而数组名不行

实际上 数组索引下标运算就是先转换成对应的指针，再通过指针去取得对应元素的

- 数组的基地址是在内存中存储数组的起始位置，数组名本身就是一个地址即指针值。
- 有效的指针运算
  - 相同类型指针的相互赋值运算
  - 指针与整数之间的加减法运算
  - 指向相同数组中元素的两个指针之间的剪发运算或比较运算
- *p++表示取得p当前指向的元素，但是p已经指向下一个元素了
- In the following declarations, the correct assignment expression is __.

  ```
  int *p[3], a[3];
  ```

  A.

  ```
  p = a
  ```

  B.

  ```
  p = &a[0]
  ```

  C.

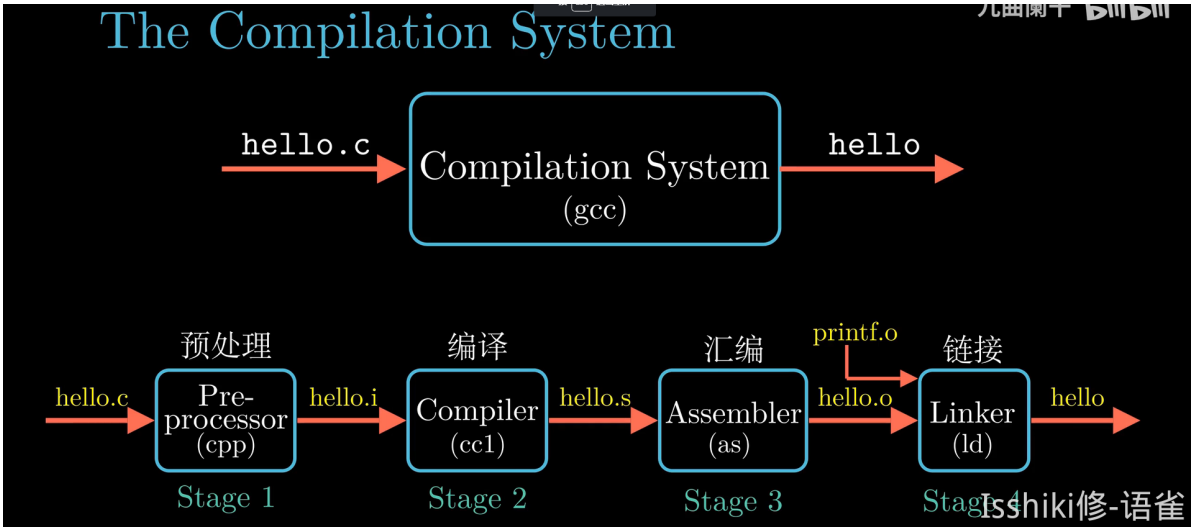
  ```
  *p = a
  ```

  D.

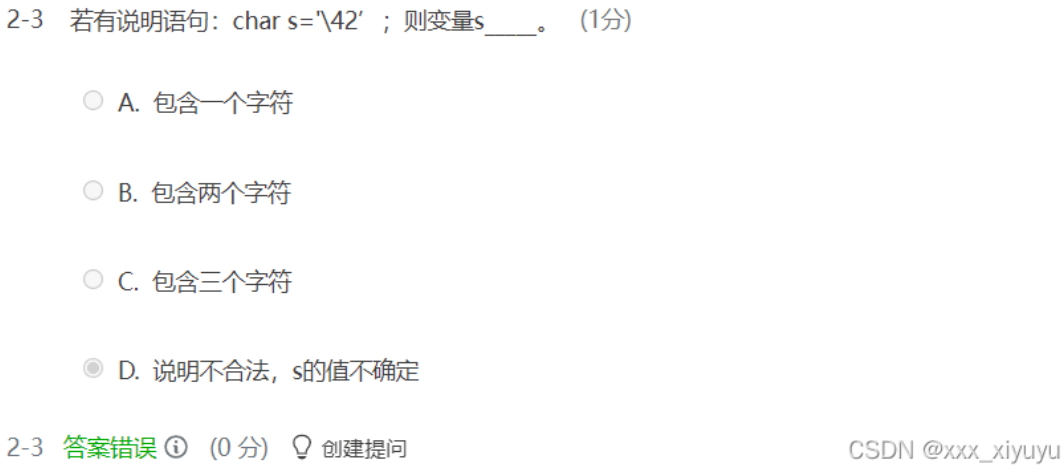
  ```
  p[0] = *a
  ```

- **问一个char型变量的值是多少，要么记得引号，要么记得用ASCII**
- 可以直接用[]访问字符串常量的某个字符
- ```
  "ABC" [1]='B'
  ```

  - `sizeof()`是一个运算符，不是函数，而且在编译阶段就完成替换。
    - 如果`k`的类型占8B内存，则`sizeof(++k)`在编译阶段被替换为8，而`k`的**值不会改变**
- `sizeof(1) = 4`
  - `sizeof(1ll) = 8`
  - `sizeof(1.0) = 8`
  - `sizeof(1.0f) = 4`
  ```

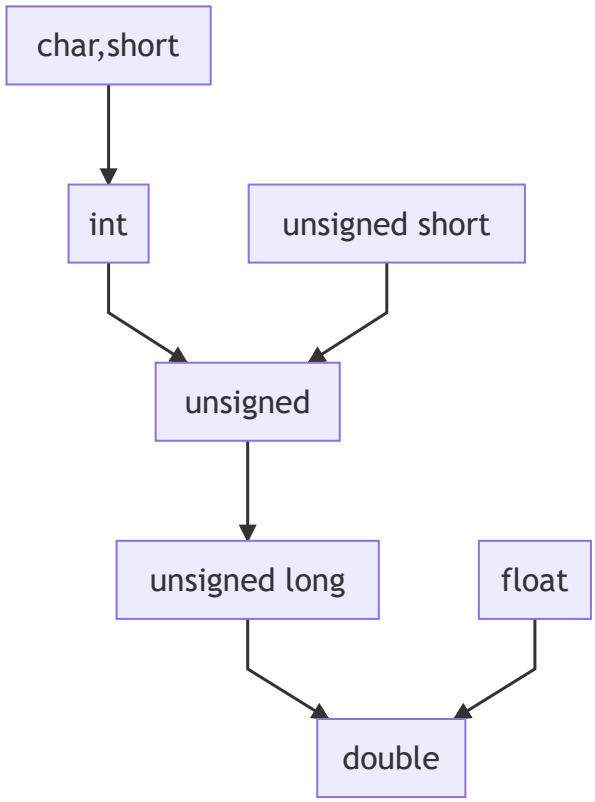


- 编译预处理是C语言编译程序的组成部分，用于解释处理C语言程序中的预处理指令（特征是以`#`开头，它们**不是**真正的C语句）
- 编译预处理在正式编译之前
- `#include`
  - 如果后面是<>，则将使用C的标准头文件
  - 如果后面是“”，将先到当前工作目录寻找被包含的文件，找不到则去系统include目录寻找
- 宏
  - 宏只是纯粹的对代码的字符串替换
- case 后跟的必须是不重复的常量表达式，不可以是变量参与的表达式
- **全局变量**的**作用域**是从**定义开始到文件结束**
- 静态局部变量的内存存储在静态存储区，生命周期持续到程序结束，不会再退出函数的时候被回收，下一次使用后会被重新激活
- 与动态全局变量相比，静态全局变量的作用域再当前文件内，动态全局变量在多文件项目中每一个文件都可以用
- 静态变量初值默认为0
- 二维数组申请时如果初始化可以省略行长度 `a[][3]={}`
- 对于二维数组`a[x][y]`
  - `&a`表示数组地址，`&a+1`表示数组末尾后的地址
  - `a`或者`&a[0]`表示数组首行地址，`a+1`表示数组下一行地址
    - `a`实际上是一个x维**数组的指针。**
  - `&a[0][0]`或者`a[0]`表示数组首元素地址，也叫列地址
  - `a[i][j]`等效于`*(*(a+i)+j)`
- 取值符号*的优先级低于成员访问符.的优先级
- 可以通过将一个字符串常量赋值给一个指针的方式来给该字符串分配地址
  - `char *p;  p="awa"`
  - `*strcpy(char *to,char *from)`函数的作用是，把**字符数组**`from[]`复制到`to[]`中并返回`to[]`
- `*strcat(char *to,char *from)`函数的作用是，把**字符数组**`from[]`添加到`to[]`末尾并返回`to[]`

- 所谓的添加，就是用`from[]`去替换`to[]`第一个`\0`之后的元素
- - `strcmp(char *a,char *b)`函数的作用是，比较**字符数组**`a[]`和`b[]`在`\0`前的内容是否等价
    - - - 如果等价 返回0
        - 如果a<b 返回-1
      - 如果a>b 返回1
- 结构体类型本身不占用内存空间，结构体变量占用内存空间
- 枚举类型中的元素都具有一个整型值
- 内存中的每一个存储单元都有一个唯一的地址
- 任何表达式语句都是表达式加分号组成的
- 文件的读函数是从输入文件中读取信息，并存放在内存中
- 文件是否打开是可以判断的
- 文件指针和位置指针都是随着文件的读写操作在不断改变❌
- 文件指针用于指向文件，文件只有被打开后才有对应的文件指针
- 一个变量的数据类型被强制转换后，它将保持被强制转换后的数据类型❌
- switch语句中，多个case可以共用一组执行语句
- 每个case常量表达式的值不可以相同
- 在同一个作用域中不可以定义同名变量，在不同的作用域中可以定义同名变量
- 指针变量不能存放数值和字符
- 字符串在内存中的起始地址称为字符串的指针
- 

- 标识符可以是字母，数字，下划线组成的字符串，并且第一个字符必须是字母或下划线。
- **\开头表示这是一个八进制转义序列，s表示一个字符**
- 标识符严格区分大小写，不能是C语言的关键字和保留标识符
- 注意if语句中是=号还是==
  - `int k=2;while(k=0){printf("%d",k);k--;}
  - 循环体语句一次也不执行
- 注释体例：`/**/`
- `x*=y+z`即为`x=x*(y+z)

- `int main()`
  { int a = 2, b = -1, c = 2;
  `if(a < b)`
  `if(b < 0)`
  `c = 0;`
  `else c++;`
  `printf("%d\n",c);` return 0;
  }
- else 总是和之前与其最近的且不带else 的if配对
- 复合语句在语法上被认为是一条语句
- 在嵌套循环中，每一层循环中都不应该改变其他层使用的循环变量的值，以免互相干扰
- `for (a=1,i=-1; -1<=i<1; i++) {`
  `a++;`
  `printf("%2d", a);}` printf("%2d", i);
  ==结果是-1==
- break 语句只能用在循环体和switch语句体内
- continue语句的作用是跳过本次循环体中余下尚未执行的语句，立即进行下一次的循环条件判定，可以理解为仅结束本次循环。
- 函数不可以嵌套定义但可以嵌套调用
- sizeof是运算符，不是函数
- C语言中，若没有对函数类型显式说明，则函数的隐含类型为int
- 函数的形参和实参分别占用不同的存储单元
- 不一定包含main函数，不能包含两个以上main函数
- `int x = 5, y = 6;` void incxy()
  `{`
  `x++;`
  `y++;}`
  `int main(void)
  `{ int x = 3;
  `incxy();` printf("%d,%d\n", x, y);
  `return 0;}`
- 3 7
- EOF实际上就是-1
- 如果`k`的类型占8B内存，则`sizeof(++k)`在编译阶段被替换为8，而`k`的**值不会改变**
- 分支语句测试数据至少需要几组问题：要注意数据边界也要算一组:
- C语言的逻辑运算具有省略特性：当前一个表达式已经能够决定整个表达式的值的时候，不计算后面那个表达式
- `int x = 0 , y = 0;` if((++x) || (++y)){
  //语句``}
  x=1,y=0
- 取余运算仅对整型数据使用
- `+ -`可作单目运算符表示正负
- 双目运算符会保证两侧的数据类型相同（对于不同的输入数据会进行自动类型转换）

- 

- 2-1 已知 `int i, a;` 执行语句 `i=(a=2*3,a*5),a+6;` 后，变量 `i` 的值是（）。 （1分）

  A. 6

  B. 12

- C. 30

  D. 36

2-1 答案错误 (0 分) 创建提问 CSDN @xxx_xiyuyu

- 1.**符号优先级**：赋值运算优先于逗号运算，2.**逗号运算**：符逗号表达式中用逗号分开的表达式分别求值，以最后一个表达式的值作为整个表达式的值。
- x为浮点型，则表达式 `x=10/4` 的值为2.0
- %运算数必须是整型
- `a[2]+3` 表示a数组行下标为2，列下标为3的元素的地址
- 连接符不能组成标识符
- 逻辑运算符两侧运算对象的数据类型可以是任何类型的数据
- C语言中所有关键字必须小写
- int long float 混合运算，结果的数据类型是double
- 再switch语句中，不一定使用break语句
- 实参与其对应的形参分别占用独立的存储单元
- continue只能运用于循环体中
- C语言源程序的扩展名是C
- C语言全局变量如果没有指定初值，则其初值自动设置为0，但局部变量不一定，局部变量如果没有指定初值，则其初值不确定

- double变量在内存中占字节数比int型变量在内存中占字节数多
- for循环的三个表达式都可以省略
- 变量被定义后，它作用域和寿命就被确定了，并且不可改变
- 条件运算符?和:是一对运算符，不能分开单独使用
- 函数的实参传递到形参有两种方式值传递和地址传递
- 若变量定义为int x, y;，则x + y = 22==不是==符合C语言语法的表达式。
- C程序中，用一对大括号{}括起来的多条语句称为复合语句，复合语句在语法上被认为是一条语句。
- !!6的值是1
- 若表达式sizeof(int)的值为4，则int类型数据可以表示的最大整数为 2^31-1(4个字节，每字节8位，一共32位，减去一个符号位)
- 与float型数据相比，double型数据的精度高，取值范围大
- 逻辑运算符两侧运算对象的数据类型可以是任意类型
- int a=4, b=3, c=2, d=1, m=1, n=3;，执行 (m=a>b>c) && (n=c>d) 后 n 的值为 （ ） 3
- 在switch语句中，每一个的case常量表达式的值==不==可以相同
- case中的变量值只能是整数常量值，不能有多余的符号
- while(i<=10);i++ ==死循环==
- 在定义数组之后，根据数组中元素的类型及个数，在内存中分配一段连续存储单元用于存放数组中的各个元素。
- 数组定义后，数组名表示该数组所分配连续内存空间中第一个单元的地址，即首地址
- 地址一般不可被更改
- 函数不能没有大括号

链表

反转链表

迭代

在遍历链表时，将当前节点的 next 指针改为指向前一个节点。由于节点没有引用其前一个节点，因此必须事先存储其前一个节点。在更改引用之前，还需要存储后一个节点。最后返回新的头引用。

```
struct ListNode* reverseList(struct ListNode* head){
    struct ListNode* prev = NULL;
    struct ListNode* curr = head;
    while(curr){
        struct ListNode* next = curr->next;
        curr->next=prev;
        prev=curr;
        curr=next;
    }
    return prev;
}
```

递归

```
struct ListNode* reverseList(struct ListNode* head){
    if(head==NULL||head->next==NULL)
    {
        return head;
    }
    struct ListNode* newHead = reverseList(head->next);
    head->next->next=head;
    head->next=NULL;
    return newHead;
}
```

回文链表

存进数组进行比较

递归

算法 `currentNode` 指针是先到尾节点，由于递归的特性再从后往前进行比较。`frontPointer` 是递归函数外的指针。若 `currentNode.val != frontPointer.val` 则返回 false。反之，`frontPointer` 向前移动并返回 true。

算法的正确性在于递归处理节点的顺序是相反的，而我们在函数外又记录了一个变量，因此从本质上，我们同时在正向和逆向迭代匹配。

```
struct ListNode* frontPointer;
bool recursivelyCheck(struct ListNode* currentNode){
    if(currentNode != NULL)
    {
        if(!recursivelyCheck(currentNode->next))
        {
            return false;
        }
        if(currentNode->val != frontPointer->val)
        {
            return false;
        }
        frontPointer = frontPointer->next;
    }
    return true;
}
bool isPalindrome(struct ListNode *head)
{
    frontPointer = head;
    return recursivelyCheck(head);
}
```

环形链表

快慢指针

假想「乌龟」和「兔子」在链表上移动，「兔子」跑得快，「乌龟」跑得慢。当「乌龟」和「兔子」从链表上的同一个节点开始移动时，如果该链表中没有环，那么「兔子」将一直处于「乌龟」的前方；如果该链表中有环，那么「兔子」会先于「乌龟」进入环，并且一直在环内移动。等到「乌龟」进入环时，由于「兔子」的速度快，它一定会在某个时刻与乌龟相遇，即套了「乌龟」若干圈。

```
bool hasCycle(struct ListNode *head) {
    struct ListNode *p,*q;
    p=head;
    q=head;
    while(q&&q->next)
    {

        p=p->next;
        q=q->next->next;
        if(p==q)
        {
            return true;
        }
    }
    return false;
}
```

相交链表

双指针法

```
struct ListNode *getIntersectionNode(struct ListNode *headA, struct ListNode
*headB) {
    if (headA == NULL || headB == NULL) {
        return NULL;
    }
    struct ListNode *pA = headA, *pB = headB;
    while (pA != pB) {
        pA = pA == NULL ? headB : pA->next;
        pB = pB == NULL ? headA : pB->next;
    }
    return pA;
}
```

链表排序

```
struct ListNode *merge(struct ListNode *head1,struct ListNode *head2)//基本的链表合
并操作
{
    struct ListNode *dummyhead = (struct ListNode*)malloc(sizeof(struct
ListNode));
    struct ListNode *temp=dummyhead,*temp1=head1,*temp2=head2;
    while(temp1&&temp2)
    {
        if(temp1->val<temp2->val)
```

```
        {
            temp->next=temp1;
            temp1=temp1->next;
        }else
        {
            temp->next=temp2;
            temp2=temp2->next;
        }
        temp=temp->next;
    }
    if(temp1) temp->next=temp1;
    if(temp2) temp->next=temp2;
    return dummyhead->next;
}
struct ListNode *sort(struct ListNode *head,struct ListNode *tail)
{
    if(head==NULL)//空链表
    return head;
    if(head->next==tail)//单节点链表
    {
        head->next=NULL;
        return head;
    }
    struct ListNode *fast=head,*slow=head;//快慢指针法寻找中点
    while(fast!=tail)
    {
        slow=slow->next;
        fast=fast->next;
        if(fast!=tail)//采用分部移动快指针的方法，防止fast移出链表
        {
            fast=fast->next;
        }
    }
    struct ListNode *mid=slow;
    return merge(sort(head,mid),sort(mid,fast));
}
struct ListNode* sortList(struct ListNode* head){
    return sort(head,NULL);
}
```

重排链表

中点反转后半段链表+合并

```
void reorderList(struct ListNode* head){
    if(head==NULL)
    return;
    struct ListNode *slow,*fast;
    slow=head,fast=head;
    while(fast&&fast->next)
    {
        slow=slow->next;
        fast=fast->next->next;
    }
```

```
    struct ListNode *l2=reverse(slow->next);
    slow->next=NULL;
    struct ListNode *l1=head;
    struct ListNode* l1_tmp;
    struct ListNode* l2_tmp;//合并链表的方法，双指针
    while (l1 && l2) {
        l1_tmp = l1->next;
        l2_tmp = l2->next;
        l1->next = l2;
        l1 = l1_tmp;
        l2->next = l1;
        l2 = l2_tmp;
    }

}
```

字符串

字符串轮转

字符串轮转。给定两个字符串`s1`和`s2`，请编写代码检查`s2`是否为`s1`旋转而成（比如，`waterbottle`是`erbottlewat`旋转后的字符串）。

`sprintf`函数

- 该函数包含在`stdio.h`头文件
- **`sprintf`函数打印到字符串中（要注意字符串的长度要足够容纳打印的内容，否则会出现内存溢出），而`printf`函数打印输出到屏幕上。**
- ```
  char str[20];
  double f=14.309948;
  sprintf(str,"%6.2f",f);//可以控制精度，数字整体长度包括小数点为6位，保留2位小数
  int a=20984,b=48090;
  sprintf(str,"%3d%6d",a,b)//将多个数值数据连接起来
  ```
- 返回值：如果成功，则返回写入的字符总数，不包括字符串追加在字符串末尾的空字符。如果失败，返回一个负数

`strstr`函数

- `strstr(str1,str2)`用于判断字符串`str2`是否是`str1`的子串
  - 如果是则该函数返回str2在str1中首次出现的地址，否则返回NULL

方法一：搜索子字符串

`s+s`包含了所有`s1`可以通过轮转操作得到的字符串
  ```

```
bool isFlipedString(char* s1, char* s2) {
    int m = strlen(s1), n = strlen(s2);
    if (m != n) {
        return false;
    }
    char * str = (char *)malloc(sizeof(char) * (m + n + 1));//注意malloc，确保
sprintf函数的实现
    sprintf(str, "%s%s", s2, s2);
    return strstr(str, s1) != NULL;
}
```

方法二：模拟

```
bool isFlipedString(char* s1, char* s2){
    int m=strlen(s1);
    int n=strlen(s2);
    if(m!=n)
    return false;
    if(n==0)
    return true;
    for(int i=0;i<n;i++)
    {
        int flag=1;
        for(int j=0;j<n;j++)
        {
            if(s2[j]!=s1[(j+i)%n])
            {
                flag=0;
            }
        }
        if(flag) return true;
    }
    return false;
}
```

字符串压缩

字符串压缩。利用字符重复出现的次数，编写一种方法，实现基本的字符串压缩功能。比如，字符串 `aabcccccaaa` 会变为 `a2b1c5a3` 。若“压缩”后的字符串没有变短，则返回原先的字符串。

```
char* compressString(char* S){//典型双指针法
   int len=strlen(S);
    if(len<=2)
        return S;
    char *str=(char*)malloc(sizeof(char)*(len*3));
    int cnt=1;
    int p=0;
    for(int i=1;i<=len;i++)
    {
        if(S[i-1]==S[i])
        {
            cnt++;
        }
```

```
        else{
            str[p++]=S[i-1];
            int wlen = sprintf(&str[p],"%d",cnt);//sprintf函数运用
            p+=wlen;//此处有易错点，cnt的位数不一样，在字符数组中占的位不同，用到sprintf函数与指针
            cnt=1;
        }
    }
    str[p]='\0';
    if(strlen(str)>=len)
        return S;
    return str;
}
```

字符串相加

给定两个字符串形式的非负整数 `num1` 和 `num2` ，计算它们的和并同样以字符串形式返回。

模拟的思想 模拟加法竖式

```
char *addStrings(char *num1,char*num2)
{
    int i=strlen(num1)-1,j=strlen(num2)-1;
    char *ans = (char*)malloc(sizeof(char)*(10000));
    int len=0,add=0;
    while(i>=0||j>=0||add!=0)
    {
        int x=i>=0?num1[i]-'0':0;
        int y=j>=0?num2[j]-'0':0;
        int result=x+y+add;
        ans[len++]='0'+result%10;
        int add=result/10;
        i--,j--;
    }
    for(int i=0;i*2<len;i++)
    {
        int t=ans[i];
        ans[i]=ans[len-1-i];
        ans[len-1-i]=t;
    }
    ans[len]='\0';
    return ans;
}
```

重新格式化字符串

给你一个混合了数字和字母的字符串 `s`，其中的字母均为小写英文字母。

请你将该字符串重新格式化，使得任意两个相邻字符的类型都不同。也就是说，字母后面应该跟着数字，而数字后面应该跟着字母。

请你返回 **重新格式化后** 的字符串；如果无法按要求重新格式化，则返回一个 **空字符串** 。

`isdigit`函数

- 头文件`#include<ctype.h>`
- 检查参数c是否为阿拉伯数字0~9，如果是，返回非零值，否则返回零

双指针法（变式）

我们把数字和字母中个数多的放在偶数位上（字符串下标从 000 开始），个数少的放在奇数位上，此时可以构造出满足题目条件的字符串。那么我们用 i 和 j 来分别表示个数多的和个数少的字符放置的下标，初始为 i=0,j=1，然后从左到右移动 iii，当 s[i] 为个数少的字符类型时，那么向右移动 j 找到往后的第一个 s[j]为个数多的字符类型，然后交换两个字符即可，不断重复该过程直至 i移动到字符串结尾即可。

```
char * reformat(char * s){
    int sum_digit = 0;
    int len = strlen(s);
    for (int i = 0; i < len; i++) {
        char c = s[i];
        if (isdigit(c)) {
            sum_digit++;
        }
    }
    int sum_alpha = len - sum_digit;
    if (abs(sum_digit - sum_alpha) > 1) {
        return "";
    }
    bool flag = sum_digit > sum_alpha;
    for (int i = 0, j = 1; i < len; i += 2) {
        if ((isdigit(s[i]) != 0) != flag) {
            while ((isdigit(s[j]) != 0) != flag) {
                j += 2;
            }
            char c = s[i];
            s[i] = s[j];
            s[j] = c;
        }
    }
    return s;
}
```

同构字符串

给定两个字符串 `s` 和 `t` ，判断它们是否是同构的。

如果 `s` 中的字符可以按某种映射关系替换得到 `t` ，那么这两个字符串是同构的。

每个出现的字符都应当映射到另一个字符，同时不改变字符的顺序。不同字符不能映射到同一个字符上，相同字符只能映射到同一个字符上，字符可以映射到自己本身。

`uthash`

- 头文件`#include<uthash.h>`

创建键—值对结构

- 每个键值对都是一个实例化的结构体
- ```
  struct  HashTable{
      int id;                     // key，可以是整型，字符，指针
      char name[10];              // value 不一定存在
      UT_hash_handle hh;          /* makes this structure hashable */
  };
  struct HashTable *users =NULL;//初始化一个哈希表，一定要初始化为NULL
  ```

HASH_FIND_INT

```
HASH_FIND_INT( users, &user_id, s ); /* s: output pointer */
//users:待查询的hash表
//&nuser_id：指向想查询的key的地址
//s：表示该函数的输出值，它是一个指向哈希表中一个键值对的指针，需要事先定义
```

```
struct my_struct *find_user(int user_id) {   /* 获得key=user_id的键值对 */
    struct my_struct *s;   /* 定义s */
    s=(struct my_struct*)malloc(sizeof(struct my_struct));
    HASH_FIND_INT( users, &user_id, s );  /* s: output pointer */
    return s;
}
```

HASH_ADD_INT

由于要保持哈希表中的唯一性，在插入键值对之前，一定要先判断表中是否已经存在要插入的键，如果已存在，就直接修改键对应的value；如果没有存在，插入键值对。

```
HASH_ADD_INT( users, id, s );  /* id: 自定义的键值对结构体中key域的变量名 ,s是待插入的键值对结构体，指针形式，key和value都要给定
```

```
void add_user(int user_id, char *name) {
    struct my_struct *s;
    HASH_FIND_INT(users, &user_id, s);  /* id already in the hash? */
    if (s==NULL) {     /* 如果s的key不存在 */
      s = (struct my_struct *)malloc(sizeof *s);
      s->id = user_id;
      HASH_ADD_INT( users, id, s );  /* id: name of key field */
    }
    strcpy(s->name, name);   /* s的key存在，直接更新value值 */
}
```

统计元素个数

```
num_numbers = HASH_COUNT(users);
```
```

循环哈希表

方法一 自己写for循环

在`uthash`中，哈希表中每个键值对之间有指针相连，并且可以通过句柄`hh`来实现指针调用。 **每个键值对都会有一个前向指针`hh.prev`与后向指针`hh.next`，因此哈希表也可以当作双向链表使用。**

```
void print_users() {
    struct my_struct *s;
    for(s=users; s != NULL; s=s->hh.next) {
        printf("user id %d: name %s\n", s->id, s->name);
    }
}
```

方法二：HASH_ITER

```
struct my_struct *s, *tmp;
HASH_ITER(hh, users, s, tmp)
//hh是句柄，s表示每次循环时获得的那个键值对，在函数前直接定义，不用赋初值，tmp,临时变量，结构体指针（不用赋值）
```

```
struct my_struct *s, *tmp;
HASH_ITER(hh, users, s, tmp) {
    printf("user id %d: name %s\n", s->id, s->name);
}
```

同构字符串解法

- 双射关系
- 维护两张哈希表，第一张哈希表s2t以s中字符为键，映射到t的字符为值，第二张则相反。从左至右遍历两个字符串的字符，不断更新两张哈希表，如果出现冲突，返回false

```
struct HashTable {
    char key;
    char val;
    UT_hash_handle hh;
};

bool isIsomorphic(char* s, char* t) {
    struct HashTable* s2t = NULL;
    struct HashTable* t2s = NULL;
    int len = strlen(s);
    for (int i = 0; i < len; ++i) {
        char x = s[i], y = t[i];
        struct HashTable *tmp1, *tmp2;
        HASH_FIND(hh, s2t, &x, sizeof(char), tmp1);
        HASH_FIND(hh, t2s, &y, sizeof(char), tmp2);
        if (tmp1 != NULL) {
            if (tmp1->val != y) {
                return false;
            }
        } else {
```

```
            tmp1 = malloc(sizeof(struct HashTable));
            tmp1->key = x;
            tmp1->val = y;
            HASH_ADD(hh, s2t, key, sizeof(char), tmp1);
        }
        if (tmp2 != NULL) {
            if (tmp2->val != x) {
                return false;
            }
        } else {
            tmp2 = malloc(sizeof(struct HashTable));
            tmp2->key = y;
            tmp2->val = x;
            HASH_ADD(hh, t2s, key, sizeof(char), tmp2);
        }
    }
    return true;
}
```

排序

快速排序

```
void qsort(int l,int r){
    int mid=a[(l+r)/2];
    int i=l,j=r;
    while(i<=j){
        while(a[i]<mid) i++;
        while(a[j]>mid) j--;
        if(i<=j){
            swap(&a[i],&a[j]);
            i++;
            j--;
        }
    }
    if(l<j) qsort(l,j);
    if(i<r) qsort(i,r);
}
```

```
void qsort(int a[],int x,int y)
{
    if(x>=y)
        return;
    int left=x;
    int right=y;
    int mid=a[left];
    while(left<=right)
    {
        while(a[right]>=mid&&left<right)
        {
            right--;
        }
        a[left]=a[right];
```

```
        while(a[left]<=mid&&left<right)
        {
            left++;
        }
        a[right]=a[left];
    }
    a[left]=mid;
    qsort(a,x,left-1);
    qsort(a,left+1,y);
}
```

插入排序

```
void insertion_sort(int a[],int len){
    int key;int i,j;
    for(i=1;i<len;i++){
        key=a[i];
        j=i-1;
        while(j>=0&&a[j]>key){
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=key;
    }
}
```

冒泡排序

```
void bubble_sort(int a[],int len)
{
    for(int i=len-1;i>0;i--)
    {
        int loc=0;
        for(int j=0;j<i;j++)
        {
            if(a[j]>a[j+1])
            {
                swap(&a[j],&a[j+1]);
                loc=j;
            }
        }
        i=loc+1;
    }
}
```

归并排序

```
void merge_sort(int a[],int start,int end)
{
    if(start>=end)
    return;
```

```
    int b[100001];
    int mid=(end+start)/2;
    int start1 = start,end1 = mid;
    int start2 = mid+1,end2 = end;
    merge_sort(a , start1 , end1);
    merge_sort(a ,  start2 , end2);
    int k=start;
    while(start1 <= end1&&start2 <= end2)
    {
        b[k++]=a[start1]<a[start2]?a[start1++]:a[start2++];
    }
    while(start1<=end1)
    {
        b[k++]=a[start1++];
    }
    while(start2<=end2)
    {
        b[k++]=a[start2++];
    }
    for(k=start;k<=end;k++)
    {
        a[k]=b[k];
    }
}
```

选择排序

```
void selection_sort(int a[],int len)
{
    for(int i=0;i<n-1;i++)
    {
        int min=i;
        for(int j=i+1;j<n;j++)
        {
            if(a[j]<a[min])
            {
                min=j;
            }
        }
        swap(&a[min],&a[i]);
    }
}
```

位运算

与运算&

- 两个位都为1时，结果才为1

> 3&5 即 0000 0011& 0000 0101 = 0000 0001，因此 3&5 的值得1。

用途

1.清零：如果想将一个单元清零，只要与一个各位都为零的数值相与，结果为零

2.取一个数的指定位：比如取数 X=1010 1110 的低4位，只需要另找一个数Y，令Y的低4位为1，其余位为0，即Y=0000 1111，然后将X与Y进行按位与运算（X&Y=0000 1110）即可得到X的指定位。

3.判断奇偶：只要根据最末位是0还是1来决定，为0就是偶数，为1就是奇数。因此可以用if ((a & 1) == 0)代替if (a % 2 == 0)来判断a是不是偶数。

或运算 |

参加运算的两个对象只要有一个为1，其值为1

用途

- 常用来对一个数据的某些位设置为1

异或运算^

```
0^0=0 0^1=1 1^0=1 1^1=0
```

参加运算的两个对象，如果两个相应位相同为0，否则为1

异或的几条性质:

- 1、交换律
- 2、结合律 (a^b)^c == a^(b^c)
- 3、对于任何数x，都有 x^x=0，x^0=x
- 4、自反性: a^b^b=a^0=a;

用途

- 翻转指定位：比如将数 X=1010 1110 的低4位进行翻转，只需要另找一个数Y，令Y的低4位为1，其余位为0，即Y=0000 1111，然后将X与Y进行异或运算（X^Y=1010 0001）即可得到。
- 与0相异或值不变
- 交换两个数

左移运算符<<

将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补0）。

设 a=1010 1110，a = a<< 2 将a的二进制位左移2位、右补0，即得a=1011 1000。

若左移时舍弃的高位不包含1，则每左移一位，相当于该数乘以2。

右移运算符>>

定义：将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。

例如：a=a>>2 将a的二进制位右移2位，左补0 或者 左补1得看被移数是正还是负。

操作数每右移一位，相当于该数除以2。

应用

快速幂

```
int quickpower(int a,int b)
{
    int ans=1;
    int base=a;
    while(b>0)
    {
        if(b&1)//b在二进制下最后一位是不是1
        {
            ans*=base;
        }
        base*=base;
        b>>=1;//b右移一位
    }
    return ans;
}
```

取余运算

$$(A+B)mod\,b=(A\,mod\,b+B\,mod\,b)mod\,b$$
$$(A\times B)\,mod\,b=((A\,mod\,b)\times(B\,mod\,b))\,mod\,b$$

```
while(b > 0)
    {
        if(b & 1)
        {
            ans *= base;
            ans %= m;
        }

        base *= base;
        base %= m;
        b >>= 1;
    }
```

位1的个数

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为[汉明重量](#)）

法一 直接计算

```
int hammingweight(int x)
{
    int cnt=0;
    for(int i=0;i<32;i++)
    {
        cnt+=(n>>i)&1;
    }
    return cnt;
}
```

法二：位运算的性质

$$n \,\&\, (n-1)$$

结果为将n二进制的最后一位1变成0

重复该操作，直到 n 的二进制表示中的全部数位都变成 00，则操作次数即为 n的位 11 的个数。

```
int hammingweight(int x)
{
    int cnt=0;
    while(x!=0)
    {
        x=x&(x-1);
        cnt++;
    }
    return cnt;
}
```

判断是不是2的整数次方

```
if(n&(n-1)==0)//2的整数次幂的二进制表示只有1个1
```