3.内存管理&编程题（20道）

3.1由gcc编译的C语言程序占用的内存分为哪几个部分?

栈区(stack)	存放函数的参数、局部变量。
堆区(heap)	提供程序员动态申请的内存空间。
全局（静态）区(static)	存放全局变量和静态变量，初始化不为0的全局变量和静态变量、const型常量在一块区域（.data段），未初始化的、初始化为0的全局变量和静态变量在相邻的另一块区域（.bss段）。
程序代码区	存放函数体的二进制代码和字符串常量。

3.2小端：一个数据的低位字节数据存储在低地址

大端：一个数据的高位字节数据存储在低地址

例如：int a=0x12345678；//a首地址为0x200，大端存储格式如下：

数据：	0x12	0x34	0x56	0x78
地址：	0x200	0x201	0x202	0x203

如何判读一个系统的大小端存储模式?

（1）方法一：int *强制类型转换为char *，用“[]”解引用

```
void checkCpuMode(void)
{
    int c = 0x12345678;
    char *p = (char *)&c;
    if(p[0] == 0x12)
        printf("Big endian.\n");
    else if(p[0] == 0x78)
        printf("Little endian.\n");
    else
        printf("Uncertain.\n");
}
```

（2）方法二：int *强制类型转换为char *，用“*”解引用

```
void checkCpuMode(void)
{
    int c = 0x12345678;
    char *p = (char *)&c;
    if(*p == 0x12)
        printf("Big endian.\n");
    else if(*p == 0x78)
        printf("Little endian.\n");
    else
        printf("Uncertain.\n");
}
```

（3）方法三：包含short跟char的共用体

```
void checkCpuMode(void)
{
    union Data
    {
        short a;
        char b[sizeof(short)];
    }data;
    data.a = 0x1234;

    if(data.b[0] == 0x12)
        printf("Big endian.\n");
    else if(data.b[0] == 0x34)
        printf("Little endian.\n");
    else
        printf("uncertain.\n");
}
```

3.3全局变量和局部变量的区别?

（1）全局变量储存在静态区，进入main函数之前就被创建，生命周期为整个源程序。

（2）局部变量在栈中分配，在函数被调用时才被创建，在函数退出时销毁，生命周期为函数内。

3.4以下程序中，主函数能否成功申请到内存空间?

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void getmemory(char *p)
{
    p = (char *)malloc(100);
    strcpy(p, "hello world");
}
int main()
{
```

```
    char *str = NULL;
    getmemory(str);
    printf("%s\n", str);
    free(str);
    return 0;
}
```

答案：不能。

解读：getmemory(str)没能改变str的值，因为传递给子函数的只是str的复制值NULL，main函数中的str一直都是 NULL。正确的getmemory()如下：

```
①传递的是二重指针，即str的指针
void getmemory(char **p)
{
    *p = (char *)malloc(100);
    strcpy(*p, "hello world");
}
②传递的是指针别名，即str的别名，C++中
void getmemory(char * &p)
{
    p = (char *)malloc(100);
    strcpy(p, "hello world");
}
```

3.5 请问运行下面的Test()函数会有什么样的后果?

```
void GetMemory(char **p, int num)
{
    *p = (char *)malloc(num);
}
void Test(void)
{
    char *str = NULL;
    GetMemory(&str, 100);
    strcpy(str, "hello");
    printf("%s\n", str);
}
```

答案：内存泄漏。

解读：调用malloc()申请内存空间，使用完毕之后没有调用free()释放内存空间并使指针指向NULL。

3.6 请问运行下面的Test()函数会有什么样的后果?

```
char *GetMemory(void)
{
    char p[] = "hello world";
    return p;
}
void Test(void)
```

```
{
    char *str = NULL;
    str = GetMemory();
    printf("%s\n", str);
}
```

答案：打印野指针内容，可能是乱码。

解读：GetMemory()返回的是指向栈内存的指针，但该栈内存已被释放，该指针的地址不是 NULL，成为野指针，新内容不可知。

3.7 请问运行下面的Test()函数会有什么样的后果?

```
void Test(void)
{
    char *str = (char *) malloc(100);
    strcpy(str,"hello");
    free(str);
    if(str != NULL)
    {
        strcpy(str, "world");
        printf("%s\n", str);
    }
}
```

答案：篡改堆区野指针指向的内容，后果难以预料，非常危险。

解读：

（1）free(str);之后，str成为野指针，没有置为NULL，if(str != NULL)语句不能阻止篡改操作。

（2）野指针不是NULL指针，是指向被释放的或者访问受限的内存的指针。

（3）造成野指针原因：①指针变量没有被初始化，任何刚创建的指针不会自动成为NULL；②指针被free或delete之后，没有置NULL；③指针操作超越了变量的作用范围，比如要返回指向栈内存的指针或引用，因为栈内存在函数结束时会被释放。

3.8在C语言中memcpy和memmove是一样的吗?

答案：

（1）memcpy()与memmove()一样都是用来拷贝src所指向内存内容前n个字节到dest所指的地址上。

（2）不同的是，当src和dest所指的内存区域重叠时，memcpy可能无法正确处理，而memmove()仍然可以正确处理，不过执行效率上略慢些。

解读：

（1）memcpy()无论什么情况下，都是从前往后拷贝内存。当源地址在前，目的地址在后，且两个区域有重叠时，会造成拷贝错误，达不到理想中的效果。

```
void *memcpy(void *dest, const void *src, size_t count)
{
    if(dest == NULL || src == NULL || count <= 0)  return NULL;
    char *d = (char *)dest;
    char *s = (char *)src;
    while(count--)
    {
        *d++ = *s++;
    }
    return dest;
}
```

（2）memmove()则分两种情况：目的地址在前，源地址在后的情况下，从前往后拷贝内容。否则从后往前拷贝内容。无论什么情况都能达到理想中的效果。

```
void *memmove(void *dest, const void *src, size_t count)
{
    if(dest == NULL || src == NULL || count <= 0)  return NULL;
    if(dest < src)
    {
        char *d = (char *)dest;
        char *s = (char *)src;
        while (count--)
        {
            *d++ = *s++;
        }
    }
    else
    {
        char *d = (char *)dest + count;
        char *s = (char *)src + count;
        while (count--)
        {
            *--d = *--s;
        }
    }
    return dest;
}
```

3.9 malloc的底层实现?

（1）malloc函数的底层实现是操作系统有一个由可用内存块连接成的空闲链表。调用malloc时，它将遍历该链表寻找足够大的内存空间，将该块一分为二（一块与用户

申请的大小相等，另一块为剩下来的碎片，会返回链表），调用free函数时，内存块重新连接回链表。

（2）若内存块过于琐碎无法满足用户需求，则操作系统会合并相邻的内存块。

3.10在1G内存的计算机中能否malloc(1.2G)？为什么?

（1）有可能。

（2）因为malloc函数是在程序的虚拟地址空间申请的内存，与物理内存没有直接的关系。虚拟地址与物理地址之间的映射是由操作系统完成的，操作系统可通过虚拟内存技术扩大内存。

3.11内存泄漏是什么?

（1）内存泄漏（Memory Leak）是指程序中已动态分配的堆内存由于某种原因未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

（2）分类：

①常发性内存泄漏：发生泄漏的代码会被多次执行到。

②偶发性内存泄漏：发生泄漏的代码在某些环境或操作下才会发生。

③一次性内存泄漏：只会被执行一次。

④隐式内存泄漏：程序在运行过程中不停地分配内存，直到结束时才释放。严格来讲这不算内存泄漏，但服务器运行时间很长，可能会耗尽所有内存。

3.12内存溢出是什么？与内存泄漏有何关系?

（1）内存溢出（Out Of Memory）是指应用系统中存在无法回收的内存或使用的内存过多，最终使得程序运行要用到的内存大于系统能提供的最大内存。此时程序无法运行，系统提示内存溢出。有时候会自动关闭软件。

（2）造成内存溢出的原因：

①内存泄漏的堆积最终导致内存溢出。

②需要保存多个耗用内存过大的对象或加载单个超大的对象时，其大小超过了当前剩余的可用内存空间。

3.13堆栈溢出一般是由什么原因导致的?

（1）堆栈溢出一般包括堆内存溢出和栈内存溢出，两者都属于缓冲区溢出。

（2）堆内存溢出可能是堆的尺寸设置得过小/动态申请的内存没有释放。

（3）栈内存溢出可能是栈的尺寸设置得过小/递归层次太深/函数调用层次过深/分配了过大的局部变量。

3.14内存溢出和内存越界的区别?

（1）内存溢出：要求分配的内存超出了系统所能给予的，于是产生溢出。

（2）内存越界：向系统申请了一块内存，而在使用时超出了申请的范围，常见的是数组访问越界。

3.15位翻转

翻转前：

数：	v8	v7	v6	v5	v4	v3	v2	v1
位：	8	7	6	5	4	3	2	1

翻转后：

数：	v1	v2	v3	v4	v5	v6	v7	v8
位：	8	7	6	5	4	3	2	1

思路：目标数初始化为0，用&0x01的方式获得原始数的第1位，然后左移7位再与目标数按位或，接着原始数右移一位；再用&0x01的方式获得原始数的第2位，然后左移6位……如此循环8次即可。最后返回目标数。

代码：

```
unsigned char bit_reverse(unsigned char input)
{
    unsigned char result = 0;
```

```
    int bit = 8;
    while(bit--)
    {
        result |= ((input & 0x01) << bit);
        input >>= 1;
    }
    return result;
}
```

3.16字符串倒序：将一个字符串的字符顺序进行前后颠倒。

思路：双指针法，两个指针，一个指向字符串开头，另一个指向字符串结尾，相互交换指向的内容，接着头指针前进，尾指针后退，交换内容……直到两指针相遇。

代码：

```
#include<string.h>
void inverted_order(char *p)
{
    char *s1, *s2, tem;
    s1 = p;
    s2 = s1 + strlen(p) - 1;
    while(s1 < s2)
    {
        tem = *s1;
        *s1 = *s2;
        *s2 = tem;
        s1++;
        s2--;
    }
}
```

3.17找出一个字符串中一个最长的连续数字，并标注出位置和长度。

思路：指针法，指针从字符串开头开始寻找数字，若找到数字，则暂时记下位置，接着不断指向下一个字符，看连续的数字有多长，直到遇到非数字字符，然后比较长度是否比上一个连续数字长，若是则记录位置跟长度，接着寻找下一个数字，直到字符串结尾。最后返回最长的连续数字的位置和长度。

代码：

```
char *find(char *a, int *size)
{
    char *in = a, *temp,*pos;
    int count = 0, max = 0;
    while(*in != '\0')
    {
        if(*in >= '0' && *in <= '9')  // 寻找数字
        {
```

```
            temp = in;
            while(*in >= '0' && *in <= '9')  // 判断长度
            {
                count += 1;
                in++;
            }
            if(count > max)  // 记录最长位置跟长度
            {
                pos = temp;
                max = count;
                count = 0;
            }
        }
        in++;
    }
    *size = max;
    return pos;
}
```

3.18写一个函数，判断输入参数是不是质数（素数）。

思路：

（1）质数是指大于1的自然数中，除了1和它本身不再有其他因数的自然数。一个大于1的自然数不是质数就是合数，因此可以将问题转换为判断合数。

（2）合数一定可以由两个自然数相乘得到，一个小于或等于它的平方根（大于1），另一个大于或等于它的平方根。因此可以判断“2 ~ 输入参数的平方根”中是否有能被输入参数整除的数，若有则该数是合数，若没有则该数是质数。

代码：

```
int IsPrime (unsigned int p)
{
    unsigned int i;
    if(p <= 1)
    {
        printf("请输入大于1的自然数。\n");
        return -1;
    }
    for(i = 2; i <= sqrt(p); i++)
    {
        if(p % i == 0)
        {
            printf("该数不是质数。\n");  // 是合数
            return 0;
        }
    }
    printf("该数是质数。\n");
    return 0;
}
```

3.19大小端转化：对一个输入的整型数进行大小端存储模式转化

思路：大小端转化就是将一个整型数的低字节放到高字节，高字节放到低字节，跟前面的位翻转类似，只不过这里的单位是字节，因此需要将位翻转中的&0x01改为&0xFF，<< bit改为size * 8，>>= 1改为 >> 8。

代码：

```
int endian_convert(int input)
{
    int result = 0;
    int size = sizeof(input);
    while(size--)
    {
        result |= ((input & 0xFF) << (size * 8));
        input >>= 8;
    }
    return result;
}
```

3.20验证回文串：给定一个字符串，验证它是否是回文串，只考虑字符和数字字符，可以忽略字母的大小写。（回文串即左右对称的字符串，如"A man, a plan, a canal: Panama")

思路：双指针法，一个指针指向字符串开头，另一个指向字符串结尾，两个指针都往中间移动并寻找字符和数字，并将字母统一转为小写，然后比较是否相同，若不相同则返回false，若相同则继续寻找……如此循环，若直到两指针相遇都没返回false，则返回true。

代码：

```
bool isPalindrome(char * s)
{
    char *left = s, *right = s + strlen(s) - 1;

    if(strlen(s) == 0) return true;

    while(left < right)
    {
        while(!
((*left >= 'a' && *left <= 'z') || (*left >= 'A' && *left <= 'Z') || (*left >= '0' && *lef
t <= '9')) && left < right)  // 找到字母或数字
            left++;
        while(!
((*right >= 'a' && *right <= 'z') || (*right >= 'A' && *right <= 'Z') || (*right >= '0' &&
 *right <= '9')) && right > left)  // 找到字母或数字
            right--;

        if(left < right)
        {
            if((*left >= 'A' && *left <= 'Z'))  // 若为大写，则转为小写
                *left = *left + 32;
            if((*right >= 'A' && *right <= 'Z'))  // 若为大写，则转为小写
```

```
                *right = *right + 32;

            if(*left == *right)  // 比较
            {
                left++;
                right--;
            }
            else
                return false;
        }
        else
            return true;
    }

    return true;
}
```

讨论 评论