# Lab 7 Part II

Lab 7 Part II 目标是对指针深入理解。

Problem 7. 指针最重要的作用是灵活地对内存进行解析，这也是学好指针的关键所在。定义一个 32 字节大小的字符数组 S，内存布局如下（表格第二行表示第 17-32 个字符）：

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ‘a’ | ‘b’ | ‘c’ | ‘ ‘ | ‘d’ | ‘e’ | ‘f’ | ‘\0’ | ‘g’ | ‘h’ | ‘i’ | ‘ ‘ | ‘j’ | ‘k’ | ‘l’ | ‘\0’ |
| ‘m’ | ‘n’ | ‘o’ | ‘ ‘ | ‘p’ | ‘q’ | ‘\0’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘ ‘ | ‘v’ | ‘\0’ | ‘x’ | ‘y’ |

编程验证以下程序的输出结果，并解释原因：

(1)

```
cout<<S<<endl;
cout<<S+1<<endl;
cout<<S+8<<endl;

cout<<&S[0]<<endl;
cout<<&S[1]<<endl;
cout<<&S[8]<<endl;
```

(2)

```
int *p = (int *)S; //将内存解释为若干个 int 类型变量
cout<<p<<endl;
cout<<*p<<endl;
cout<<p[0]<<endl;
cout<<*(p+1)<<endl;
cout<<p[1]<<endl;
cout<<*p + 1<<endl;
```

(3)

```
float *p = (float *)S; //将内存解释为若干个 float 类型变量
cout<<p<<endl;
cout<<*p<<endl;
cout<<p[0]<<endl;
cout<<*(p+1)<<endl;
cout<<p[1]<<end;
cout<<*p + 1<<endl;
```

(4)

```
double *p = (double *)S; //将内存解释为若干个 double 类型变量
cout<<p<<endl;
cout<<*p<<endl;
cout<<*(p+1)<<endl;
cout<<*p + 1<<endl;
```

(5)

```
unsigned int (*p) [2] = (unsigned int (*)[2])S; //将内存解释为若干个 unsigned int [2]类型的数组
cout<<p<<endl;
cout<<*p<<endl;
cout<<*(p+1)<<endl;
cout<<**p<<endl;
cout<<*(*p+1)<<endl;
cout<<**(p+1)<<endl;
cout<<p[0]<<endl;
cout<<p[0][0]<<endl;
cout<<p[1][0]<<endl;
cout<<p[0][1]<<endl;
```

(6)

```
char (*p)[8] = (char (*)[8])S; //将内存解释为若干个 char [8]类型的数组
cout<<p<<endl;
cout<<*p<<endl;
cout<<*(p+1)<<endl;
```

(7)

```
unsigned short (*p) [2][2] = (unsigned short (*) [2][2])S; //将内存解释为若干个 unsigned short [2][2]类型的数组
cout<<p<<endl;
cout<<*p<<endl;
cout<<*(p+1)<<endl;
cout<<**p<<endl;
cout<<*(*p+1)<<endl;
cout<<**(p+1)<<endl;
cout<<***p<<endl;
cout<<***(p+1)<<endl;
cout<<**(*p + 1)<<endl;
cout<<*(**p + 1)<<endl;
cout<<p[0]<<endl;
cout<<p[0][0]<<endl;
cout<<p[1][0]<<endl;
cout<<p[0][1]<<endl;
cout<<p[0][0][0]<<endl;
cout<<p[0][1][0]<<endl;
```

Problem 8. 动态分配内存。

(1) 从键盘输入整数 n（n 为变量），动态分配大小为 n 的 int 类型数组，从键盘输入 n 个整数，排序后输出。

(2) 从键盘输入整数 n（n 为变量），动态分配大小为 n 的字符型数组，从键盘输入字符串（长度小于 n），将字符串反序输出。

(3) 从键盘输入整数 n 和 m（m 和 n 都是变量），表示一个矩阵的行和列。从键盘输入 n 行数据，每行包含 m 个整数，空格隔开，代表矩阵元素。使用动态分配内存存储该矩阵，并将矩阵转置后输出（提示：先动态生成一维指针数组，每个指针再赋值为动态一维数组）。

Problem 9. 复现 memcpy。
memcpy 的功能是将一段内存的内容复制到另一段内存，例如，将一个长度为 16 字节的 char 型数组的内容复制到一个有 4 个元素的 int 型数组。根据下面函数原型，实现 memcpy 函数。
void mymemcpy(void *src, void *dest, int size) {
//src 表示源内存地址，desti 表示目标内存地址，size 表示要拷贝的内存大小
}

Problem 10. 函数指针。
理解以下概念
你到一个商店买东西，刚好你要的东西没有货，于是你在店员那里留下了你的电话，过了几天店里有货了，店员就打了你的电话，然后你接到电话后就到店里去取了货。在这个例子里，你的电话号码就叫回调函数，你把电话留给店员就叫登记回调函数，店里后来有货了叫做触发了回调关联的事件，店员给你打电话叫做调用回调函数，你到店里去取货叫做响应回调事件。

(1) 你有两个函数指针 p1,p2 他们的返回值都是 int，参数列表都是(int a,int b)
实现一个函数 int sum(int order,int a,int b, ???(自己完善后半部分) )
在 order 等于 1 的时候调用p1 返回 p1(a,b)的值，否则调用p2 返回 p2(a,b)的值

(2) 实现一个指针数组，里面放着很多函数的指针，依次为把一个 int 数字平方，立方，四次方，五次方，等等。在 main 函数中使用这些函数指针。

(3)实现 map 函数，这个函数的输入是一个函数指针和一个数组指针，功能是对于数组的所有元素依次调用该函数，具体来说实现以下的功能：

```
int haha(int x){ return x*x;}
int a[6]={0,1,2,3,4,5};
for(int i=0;i<6;i++){
    a[i]=haha(a[i]);
}
```

(3) 指针会引起安全问题，下面的程序就是一个例子。

```
#include <stdio.h>
#include <string.h>
void fail(){}; //空函数
void win()//函数，输出一个字符串
```

```
{
    cout<<"Congratulations, you pwned it.\n";
}
int main()
{
    void (* fp) (); //定义函数指针
    char buffer[4];
    fp = fail; //让 fp 指向 fail 函数
    gets(buffer); //输入一个字符串
    if ((int)fp==0x30303030)//如果 fp 的值等于 0x30303030，将调用 win 函数
    {

        win(); // 调用 win

    }
    return 0;
}
```

已知 fail 函数的地址并不是 0x30303030，但我们发现，该函数在用户输入某个字符串的时候，却引发了 win 函数的调用，即 fp == 0x30303030 在某个输入的情况下成立了。你知道用户输入了什么吗？（提示：gets 函数会引起缓冲区溢出，如果用户输入了长度大于字符数组的字符串，其他内存位置的内容，比如变量 fp，就会被覆盖）

Problem 11.

内存地址分为大端编码（比如 int 变量占 4 各字节，所表示的整数的低位字节在高地址端）和小端编码（低位字节在低地址端）。写一个函数输出你目前的机器是大端编码还是小端编码。