指针、函数、预处理器

1、指针

> 指针是一个变量，其值为地址。
>
> 声明指针或者不再使用后都要将其置为0 (NULL)
>
> 野指针 未初始化的指针
>
> 悬空指针 指针最初指向的内存已经被释放了的一种指针

```
int *a; 正规
int* a;
int * a;
//因为 其他写法看起来有歧义
int* a,b;
```

使用:

```
//声明一个整型变量
int i = 10;
//将i的地址使用取地址符给p指针
int *p = &i;
//输出 0xffff 16进制地址
printf("%#x\n", &i);
printf("%#x\n", &p);
```

> 指针多少个字节？指向地址，存放的是地址

> 地址在 32位中指针占用4字节 64为8

```
//32位:
sizeof(p) == 4;
//64位:
sizeof(p) == 8;
```

解引用

> 解析并返回内存地址中保存的值

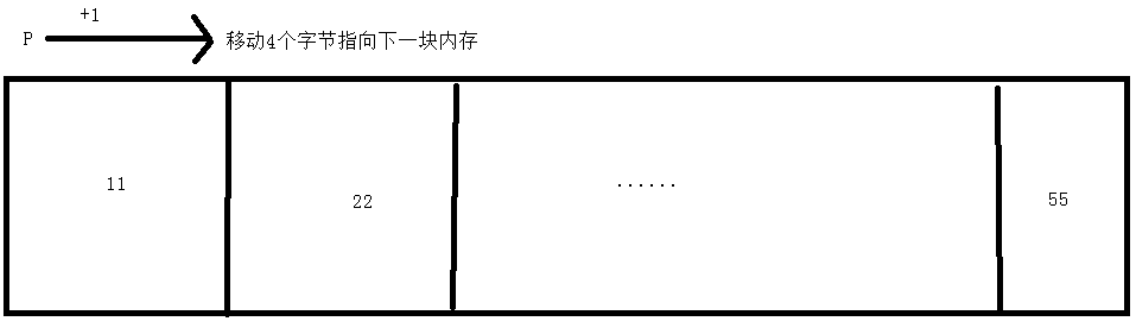
```
int i = 10;
int *p = &i;
//解引用
//p指向一个内存地址，使用*解出这个地址的值 即为 10
int pv = *p;
//修改地址的值,则i值也变成100
//为解引用的结果赋值也就是为指针所指的内存赋值
*p = 100;
```

指针运算

```
//对指针 进行算数运算
//数组是一块连续内存 分别保存 11-55
//*p1 指向第一个数据 11，移动指针就指向第二个了
int i1[] = {11,22,33,44,55};
int *p1 = i1;
for (size_t i = 0; i < 5; i++)
{
    //自增++ 运算符比 解引用* 高,但++在后为先用后加
    //如果++在前则输出 22-55+xx
    printf("%d\n", *p1++);
    //p1[0] == *(p1+1) == s[1]

}
```

> 指针指向地址，指针运算实际上就是移动指针指向的地址位置,移动的位数取决于指针类型（int就是32位)



数组和指针

> 在c语言中，指针和数组名都表示地址
>
> 1、数组是一块内存连续的数据。

> 2、指针是一个指向内存空间的变量

```
int i1[] = {11,22,33,44,55};
//直接输出数组名会得到数组首元素的地址
printf("%#x\n",i1);
//解引用
printf("%d\n",*i1);
//将数组名赋值给一个指针，这时候指针指向数组首元素地址
int *p1 = i1;

//数组指针
//二维数组类型是 int (*p)[x]
int array[2][3] = { {11,22,33},{44,55,66} };
//也可以 int array[2][3] = {  11,22,33 ,44,55,66 };
//array1 就是一个 int[3] 类型的指针
int (*array1)[3] = array;
//怎么取 55 ？
//通过下标
array[1][1] == array1[1][1]
//通过解引用
int i = *(*(array1 + 1) + 1);
//拆分
//1、 指针偏移 因为array1的类型是3个int的数组 所以 +1 移动了12位
array1 + 1
//2、获得{44,55,66}数组  (*(array1 + 1))[1] = 55
(*(array1 + 1)
//3、对数组执行 +/- 相当于隐式的转为指针
//获得 55 的地址 再解地址
*(array1 + 1) + 1


//指针数组
int *array2[2];
array2[0] = &i;
array2[1] = &j;
```

const char *, char const *, char * const，char const * const

> const： 常量 = final

```
//从右往左读
//P是一个指针 指向 const char类型
char str[] = "hello";
const char *p = str;
str[0] = 'c'; //正确
p[0] = 'c';   //错误 不能通过指针修改 const char

//可以修改指针指向的数据
//意思就是 p 原来 指向david，
//不能修改david爱去天之道的属性，
//但是可以让p指向lance，lance不去天之道的。
p = "12345";

//性质和 const char * 一样
char const *p1;
```

```
//p2是一个const指针 指向char类型数据
char * const p2 = str;
p2[0] = 'd';  //正确
p2 = "12345"; //错误

//p3是一个const的指针变量 意味着不能修改它的指向
//同时指向一个 const char 类型 意味着不能修改它指向的字符
//集合了 const char * 与  char * const
char const* const p3 = str;
```

多级指针

> 指向指针的指针
>
> 一个指针包含一个变量的地址。当我们定义一个指向指针的指针时，第一个指针包含了第二个指针的地址，第二个指针指向包含实际值的位置。

```
int a = 10;
int *i = &a;
int **j = &i;
// *j 解出 i
printf("%d\n", **j);
```

多级指针的意义

> 见函数部分的引用传值

2、函数

> C中的函数与java没有区别。都是一组一起执行一个任务的语句，也都由 **函数头**与**函数体**构成

函数的位置

> 声明在使用之前

函数参数

传值调用

把参数的值复制给函数的形式参数。修改形参不会影响实参

引用调用

形参为指向实参地址的指针，可以通过指针修改实参。

```
void change1(int *i) {
    *i = 10;
}
void change2(int *i) {
    *i = 10;
}
int i = 1;
change1(i);
printf("%d\n",i); //i == 1
change2(&i);
printf("%d\n",i); //i == 10
```

可变参数

> 与Java一样，C当中也有可变参数

```
#include <stdarg.h>
int add(int num, ...)
{
    va_list valist;
    int sum = 0;
    // 初始化  valist指向第一个可变参数 (...)
    va_start(valist, num);
    for (size_t i = 0; i < num; i++)
    {
        //访问所有赋给 valist 的参数
        int j = va_arg(valist, int);
        printf("%d\n", j);
        sum += j;
    }
    //清理为 valist 内存
    va_end(valist);
    return sum;
}
```

函数指针

> 函数指针是指向函数的指针变量

```
void println(char *buffer) {
    printf("%s\n", buffer);
}
//接受一个函数作为参数
void say(void(*p)(char*), char *buffer) {
    p(buffer);
}
void(*p)(char*) = println;
p("hello");
//传递参数
say(println, "hello");

//typedef 创建别名 由编译器执行解释
//typedef unsigned char u_char;
typedef void(*Fun)(char *);
```

```
Fun fun = println;
fun("hello");
say(fun, "hello");

//类似java的回调函数
typedef void(*Callback)(int);

void test(Callback callback) {
    callback("成功");
    callback("失败");
}
void callback(char *msg) {
    printf("%s\n", msg);
}

test(callback);
```

3、预处理器

> 预处理器不是编译器，但是它是编译过程中一个单独的步骤。
>
> 预处理器是一个文本替换工具
>
> 所有的预处理器命令都是以井号（#）开头

常用预处理器

预处理器	说明
#include	导入头文件
#if	if
#elif	else if
#else	else
#endif	结束 if
#define	宏定义
#ifdef	如果定义了宏
#ifndef	如果未定义宏
#undef	取消宏定义

宏

预处理器是一个文本替换工具

> 宏就是文本替换

```
//宏一般使用大写区分
//宏变量
//在代码中使用 A 就会被替换为1
#define A 1
//宏函数
```

```
#defind test(i) i > 10 ? 1: 0

//其他技巧
// # 连接符 连接两个符号组成新符号
#define DN_INT(arg) int dn_ ## arg
DN_INT(i) = 10;
dn_i = 100;

// \ 换行符
#define PRINT_I(arg) if(arg) { \
 printf("%d\n",arg); \
 }
PRINT_I(dn_i);

//可变宏
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR,"NDK", __VA_ARGS__);

//陷阱
#define MULTI(x,y)  x*y
//获得 4
printf("%d\n", MULTI(2, 2));
//获得 1+1*2  = 3
printf("%d\n", MULTI(1+1, 2));
```

> 宏函数
>
> 优点:
>
> 文本替换，每个使用到的地方都会替换为宏定义。
>
> 不会造成函数调用的开销（开辟栈空间，记录返回地址，将形参压栈，从函数返回还要释放堆
>
> 栈。）
>
> 缺点:
>
> 生成的目标文件大，不会执行代码检查
>
> 内联函数
>
> 和宏函数工作模式相似，但是两个不同的概念，首先是函数，那么就会有类型检查同时也可以 debug 在编译时候将内联函数插入。
>
> 不能包含复杂的控制语句，while、switch，并且内联函数本身不能直接调用自身。 如果内联函数的函数体过大，编译器会自动的把这个内联函数变成普通函数。

作业:手写sprintf

根据可变参数、指针运算等知识自己实现 sprintf 函数（只实现 %d 就行）！