Easy搞定

恶魔C语言

(提高篇)

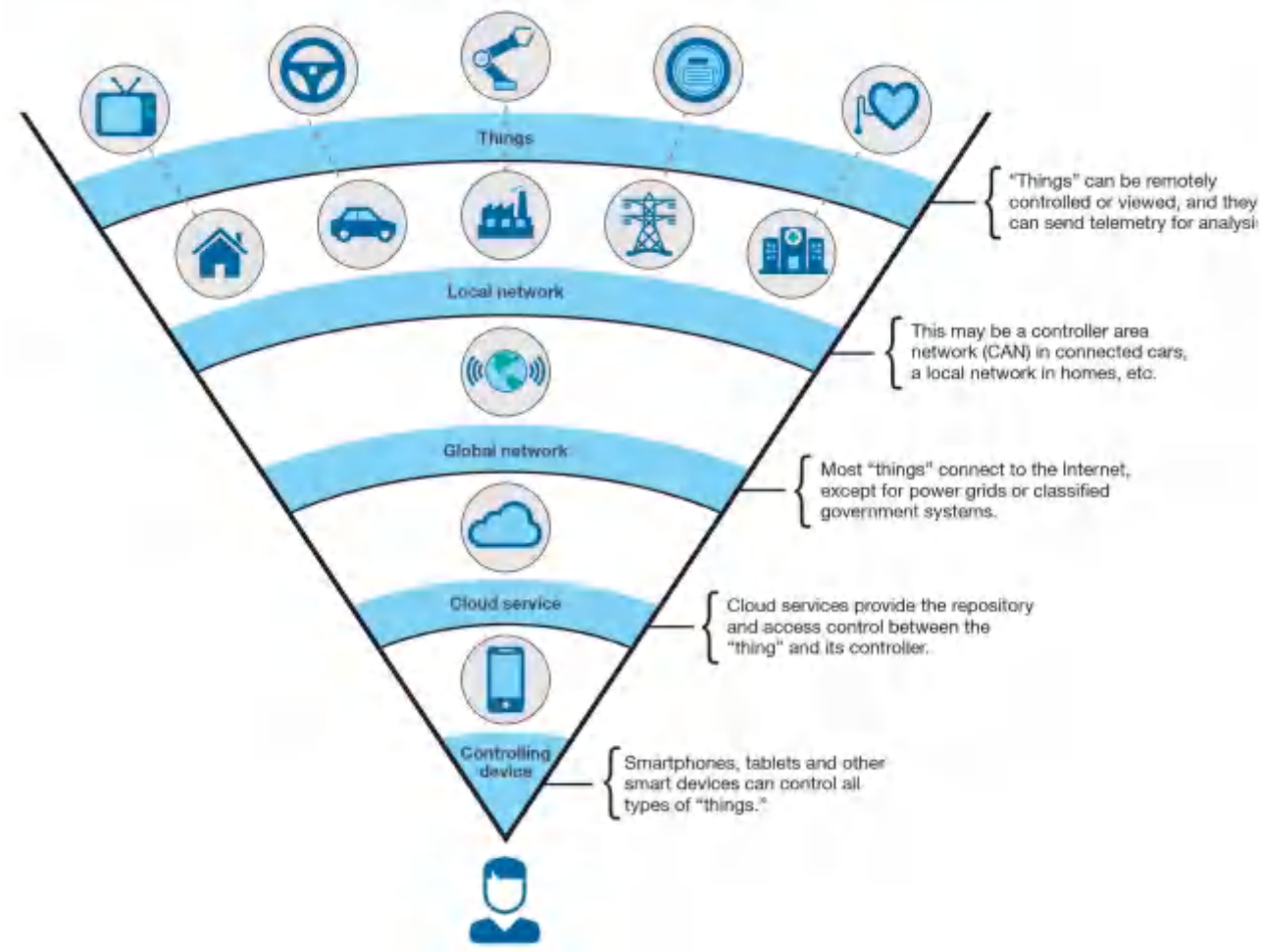
Make C be an Angel from demon

传智播客 无崖子

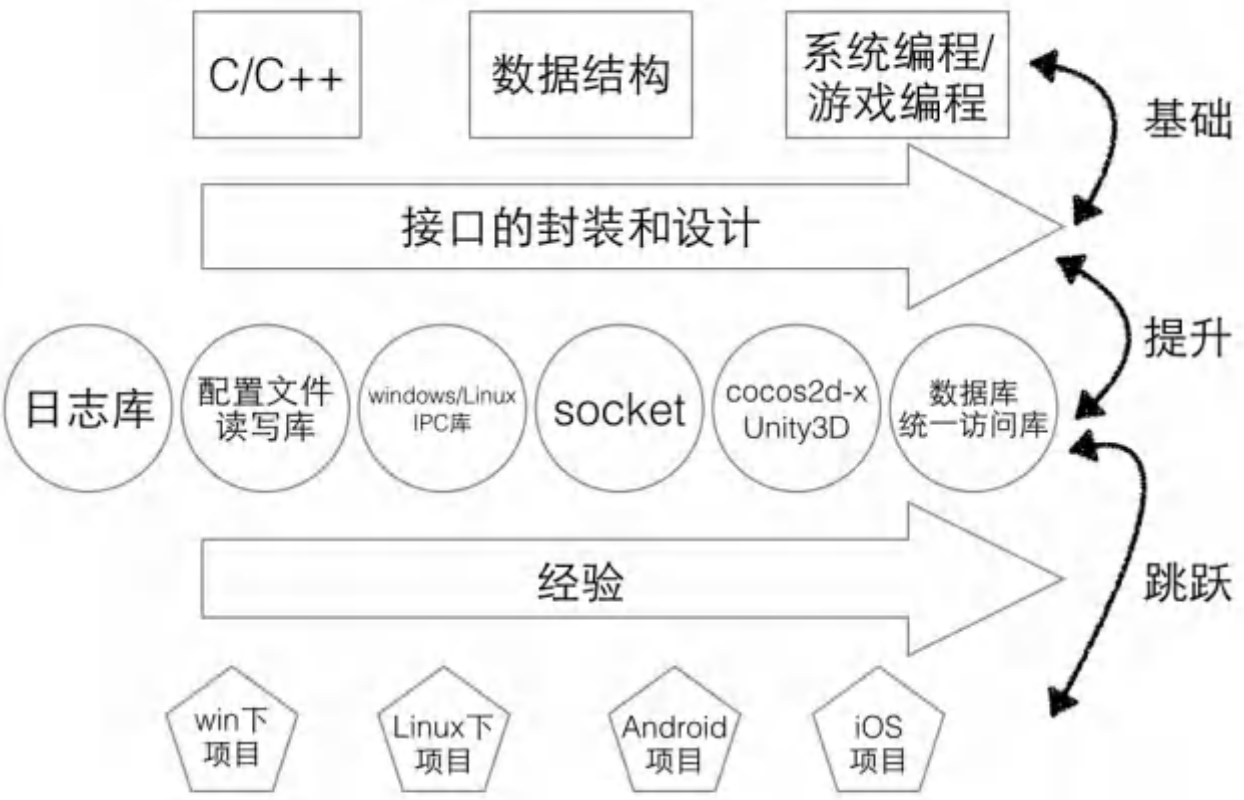
0 精神与契约

企业需要能干活的人，需要能上战场的兵。

0.1 如何成为一个对企业有价值的人?



对于解决方案有很清晰的架构图，
那么对于我们的技术也要分清层次



0.2 上战场的能力

0.2.1 过程的封装设计

0.2.1 接口的封装和设计(功能的抽象和封装)

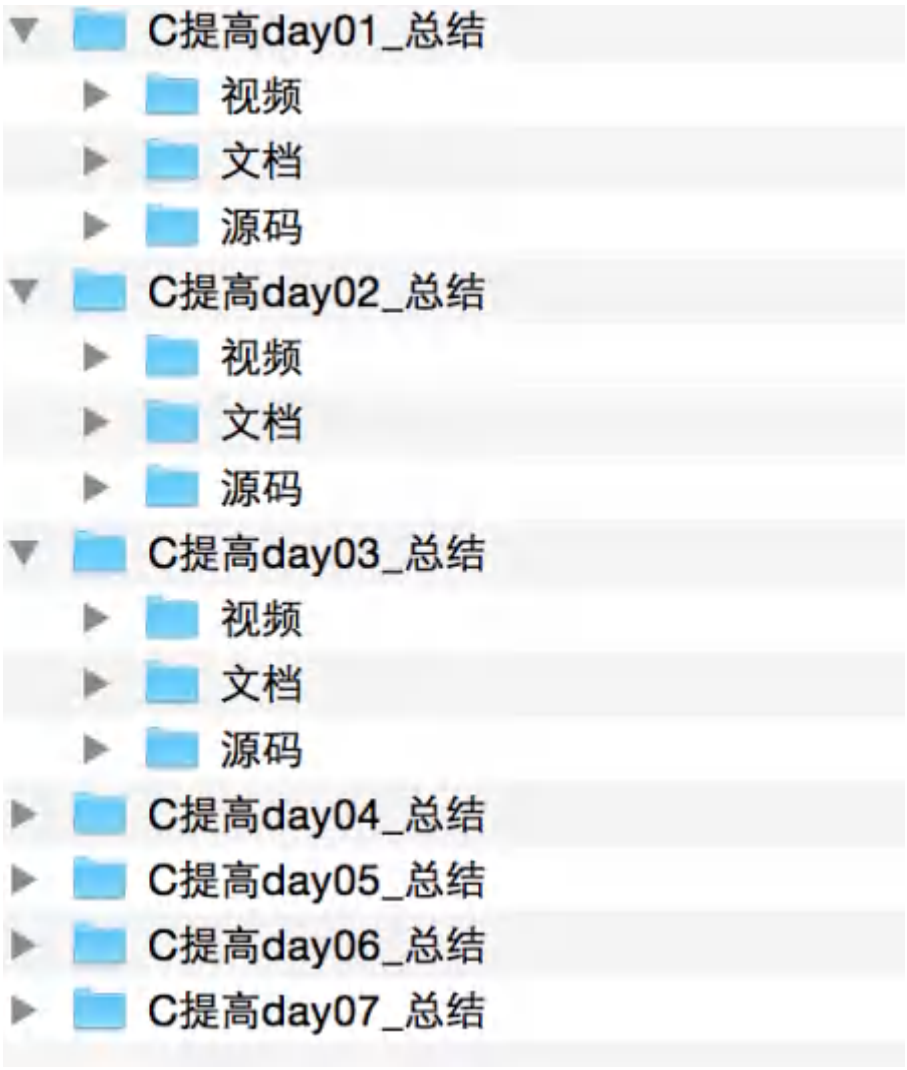
- 接口api的使用能力
- 接口api的查找能力
- 接口api的实现能力

0.2.2 建立正确程序运行内存布局图

- 内存四区模型
- 函数调用模型

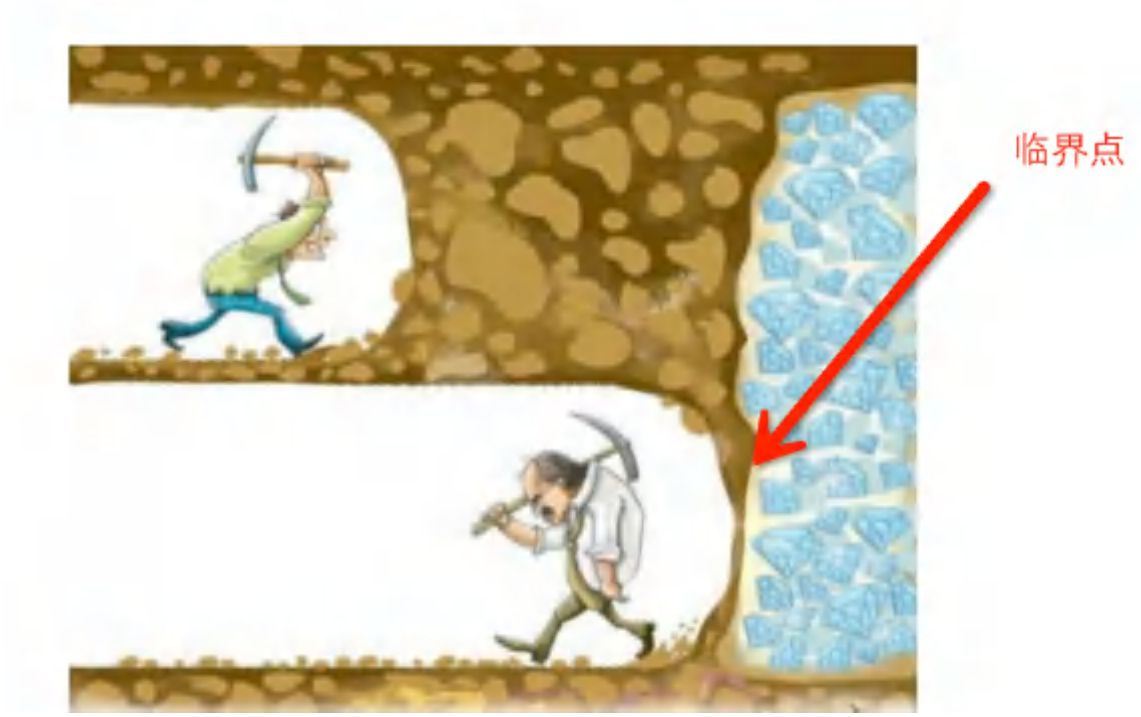
0.3 战前准备:

- 资料管理



- 工作经验，记录和积累
- 临界点

当你感觉很吃力的东西，往往是对你最有价值的东西。



既然你已经确定了方向，就不要再犹豫。


伦敦某条街
卖火柴
卖火柴
卖火柴
vitamincool@hanmail.net
啊....好冷.....
一个都没卖掉
呼呼~
呼呼~
我的人生怎么
这么辛苦.....


孩子，
给我一盒火柴！
啊！
好！！
但是钱太大了
找不开……
不用找了


不要放弃~
坚持下去
10年后会有成就的
知道了~叔叔！
再辛苦我也会
坚持下去的！

做一件事，一定要坚持。

- 当堂运行
- 动手
- 看问题的角度

- 课堂

  专心听讲、积极思考
  遇到不懂的暂时先记下,课后再问
  建议准备一个笔记本(记录重点、走神的时间)
  杜绝边听边敲、杜绝犯困听课

- 课后

  从笔记、代码等资料中复习上课讲过的知识点
  如果时间允许,请前做好预习

尽量少回看视频,别对视频产生依赖,可以用2倍速度回看视频 按时完成老师布置的练习,记录练习中遇到的BUG和解决方案 根据自己的理解总结学到的知识点

初学者应该抓住重点,不要钻牛角尖 遇到问题了,优先自己尝试解决,其次谷歌百度,最后再问老师 时间允许,可以多去网上找对应阶段的学习资料面试题,注意作息,积极锻炼。

0.4 教学思想和目标

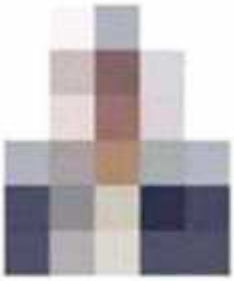
教学思想

理论 + 实践。 实践>理论 先抛出问题,再引出解决方案(知识点)
授人以鱼,不如授之以渔 。

教学目标

累积上10万行的代码量 能够独立解决开发过程中80%的BUG,成为优秀的“BUG终结者” 。

小学老师



初中老师

高中老师

大学老师

曾经的我，
只记得认识一位老师。

苍老师

1. 规划内存四区领地

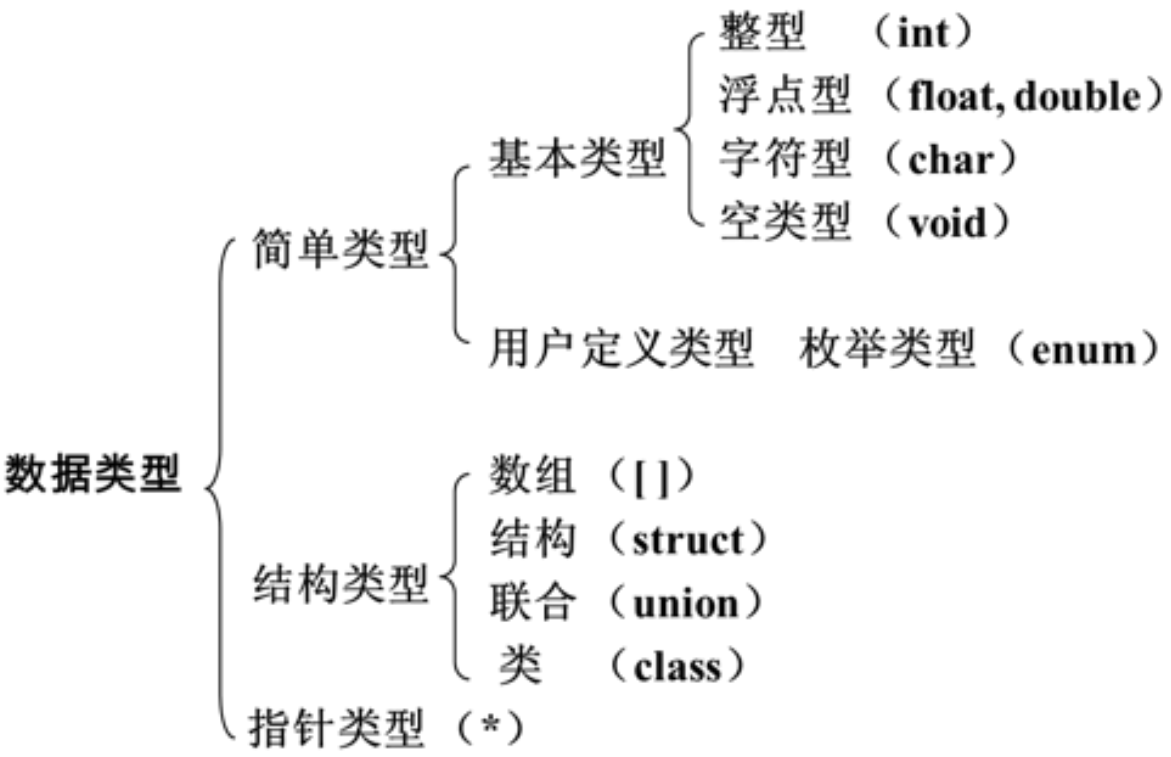
我们还面临哪些问题要解决？

数据类型的引申和思考
变量的本质
内存的操作

1.1 数据类型本质分析

1.1.1 数据类型概念

- “类型”是对数据的抽象
- 类型相同的数据有相同的表示形式、存储格式以及相关的操作
- 程序中使用的所有数据都必定属于某一种数据类型



数据类型的本质思考

- 思考数据类型和内存有关系吗？
- C/C++为什么会引入数据类型？

从编译器的角度来考虑数据类型问题，才会发现他的本质。

1.1.2 数据类型的本质

- 数据类型可理解为创建变量的模具；是固定内存大小的别名。
- 数据类型的作用：编译器预算对象（变量）分配的内存空间大小

```c
#include <stdio.h>

int main(void)
{
    int a = 10; //告诉编译器，分配4个字节的内存
    int b[10];  //告诉编译器，分配4*10 = 40 个字节的内存

    printf("b:%p, b+1: %p, &b:%p, &b+1: %p\n", b, b + 1, &b, &b + 1);

    //b+1 和 &b+1的结果不一样
    //是因为 b 和 &b 所代表的数据类型不一样
    //b  代表数组首元素的地址
    //&b 代表整体数组的地址

    return 0;
}
```

1.1.3 数据类型的大小

```c
#include <stdio.h>

int main(void)
{
    int a = 10; //告诉编译器，分配4个字节的内存
    int b[10];  //告诉编译器，分配4*10 = 40 个字节的内存

    printf("sizeof(a):%d \n", sizeof(a));
    printf("sizeof(int *):%d \n", sizeof(int *));
    printf("sizeof(b):%d \n", sizeof(b));
```

```
    printf("sizeof(b[0]):%d \n", sizeof(b[0]));
    printf("sizeof(*b):%d \n", sizeof(*b));
    return 0;
}
```

sizeof是操作符，不是函数；sizeof测量的实体大小为编译期间就已确定。

1.1.4 数据类型的别名

```
#include <stdio.h>

struct People
{
    char name[64];
    int age;
} ;

typedef struct People
{
    char name[64];
    int age;
} people_t;
/* 给结构体类型起别名 */

typedef unsigned int u32;   //给unsigned int类型取别名

int main(void)
{
    struct People p1;
    people_t p2;
    u32 a;

    p1.age = 10;
    p2.age = 11;

    a = 10;

    return 0;
}
```

1.1.5 数据类型的封装

- void的字面意思是“无类型”，void *则为“无类型指针”，void *可以指向任何类型的数据。

用法1：数据类型的封装。

```
int InitHardEnv(void **handle);
```

典型的如内存操作函数memcpy和memset的函数原型分别为

```
void * memcpy(void *dest, const void *src, size_t len);
void * memset ( void * buffer, int c, size_t num );
```

用法2： void修饰函数返回值和参数，仅表示无。

如果函数没有返回值，那么应该将其声明为void型
如果函数没有参数，应该声明其参数为void

```
int function(void)
{
    return 1;
}

void function2(void)
{
    return;
}
```

- void指针的意义

C语言规定只有相同类型的指针才可以相互赋值

void*指针作为左值用于“接收”任意类型的指针
void*指针作为右值赋值给其它指针时需要强制类型转换

```
int *p1 = NULL;
char *p2 = (char *)malloc(sizoeof(char)*20);
```

- 不存在void类型的变量

C语言没有定义void究竟是多大内存的别名.

1.1.6 数据类型的总结与拓展

1、数据类型本质是固定内存大小的别名；是个模具，c语言规定：通过数据类型定义变量。

2、数据类型大小计算（sizeof）

3、可以给已存在的数据类型起别名typedef

4、数据类型封装概念（void 万能类型）

思考1：

C语言中一维数组、二维数组有数据类型吗？int array[10]。
a)若有，数组类型又如何表达？又如定义？
b)若没有，也请说明原因。

初学者需要征服的三座大山

1、数组类型

2、数组指针

3、数组类型和数组指针的关系

思考2：

C语言中，函数是可以看做一种数据类型吗？
a)若是，请说明原因
并进一步思考：函数这种数据类型，能再重定义吗？
b)若不是，也请说明原因。

1.2 变量的本质分析

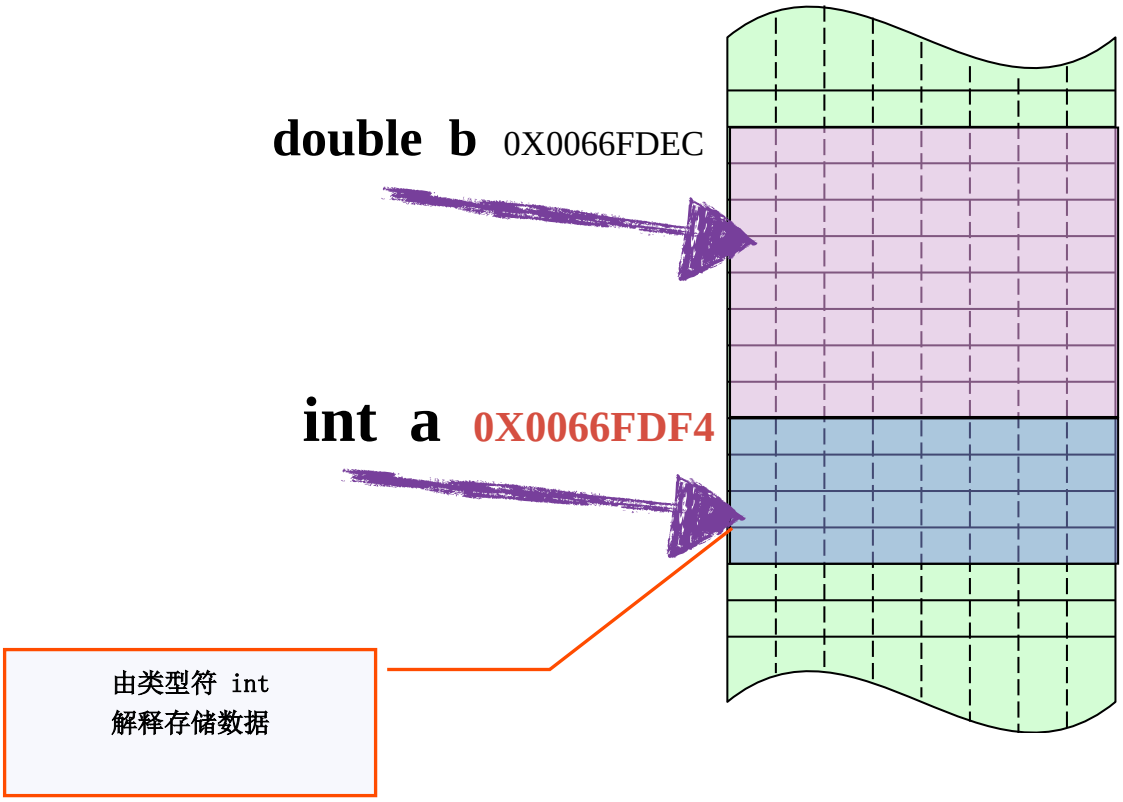
1.2.1 变量的概念

概念：既能读又能写的内存对象，称为变量；若一旦初始化后不能修改的对象则称为常量。

变量定义形式：　类型 标识符, 标识符, … , 标识符；

```
int       x;
int       wordCut,  Radius,  Height;
double    FlightTime,  Mileage,  Speed;
```


```
int a;
double b;
```



1.2.2 变量的本质

1、程序通过变量来申请和命名内存空间 int a = 0。

2、通过变量名访问内存空间。

> 变量：一段连续内存空间的别名.

3、修改变量有几种方法？

1）、直接.
2）、间接。内存有地址编号，拿到地址编号也可以修改内存；

```c
#include <stdio.h>

int main(void)
{
    int i = 0;

    // 通过变量直接操作内存
    i = 10;

    // 通过内存编号操作内存
    printf("&i:%d\n", &i);      //1245024

    *((int *)(1245024)) = 100;
    printf("i:%d", i);

    return 0;
}
```

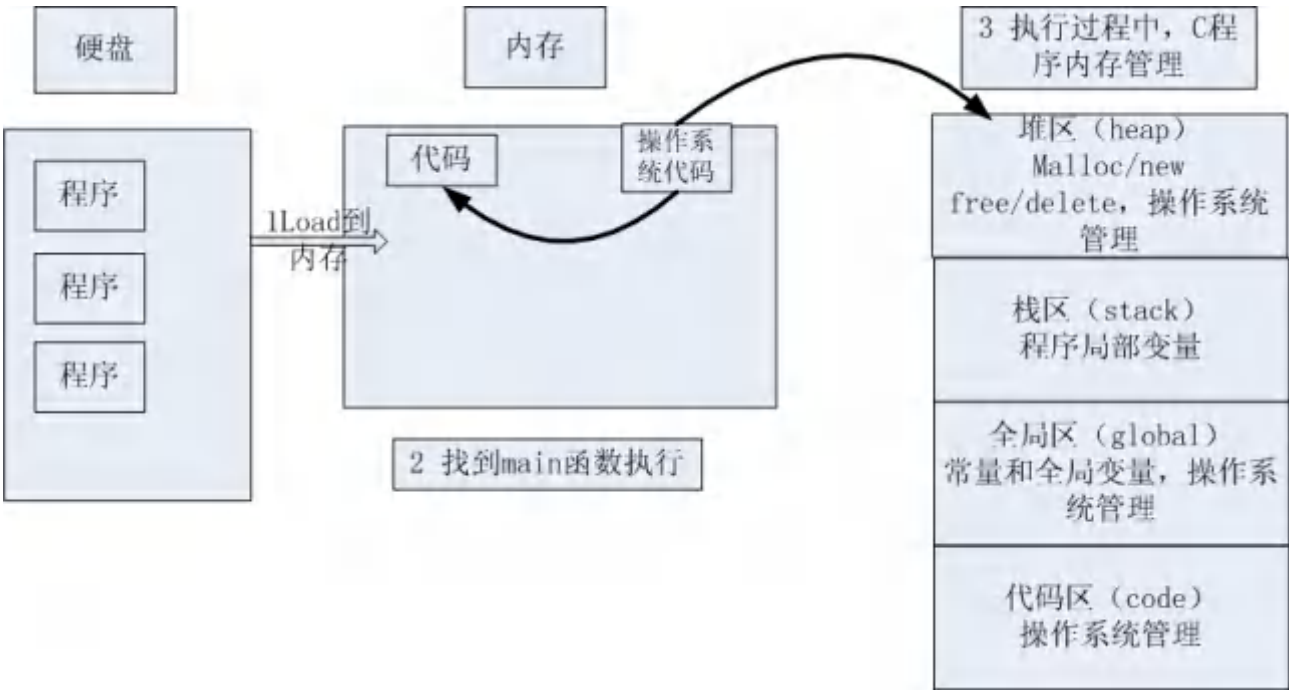
4、数据类型和变量的关系

> 通过数据类型定义变量

5、总结

> 1 对内存，可读可写；
> 2 通过变量注内存读写数据；
> 3 不是向变量读写数据，而是向变量所代表的内存空间中写数据。

1.3 程序的内存四区模型



流程说明

1、操作系统把物理硬盘代码load到内存

2、操作系统把c代码分成四个区

3、操作系统找到main函数入口执行

栈区（stack）：由编译器自动分配释放，存放函数的参数值，局部变量的值等。
堆区（heap）：一般由程序员分配释放（动态内存申请与释放），若程序员不释放，程序结束时可能由操作系统回收。
全局区（静态区）（static）：全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域，该区域在程序结束后由操作系统释放。
常量区：字符串常量和其他常量的存储位置，程序结束后由操作系统释放。
程序代码区：存放函数体的二进制代码。

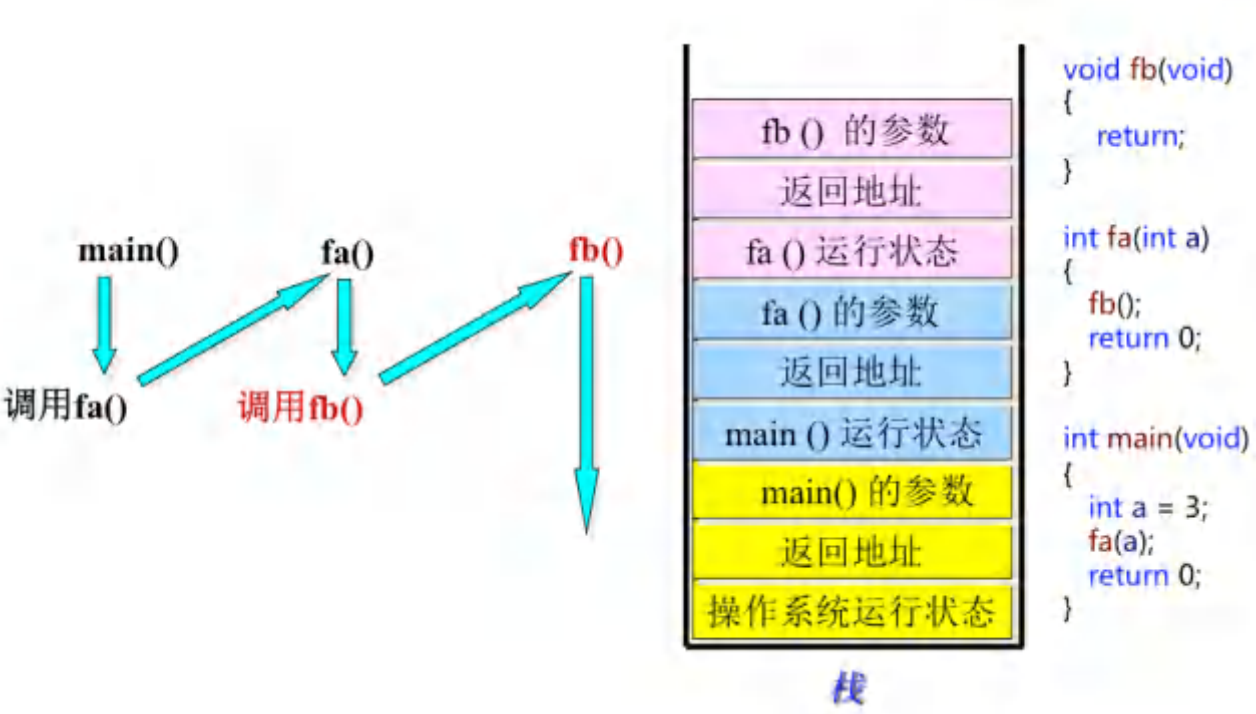
1.3.1 栈区和堆区

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//堆
char *getMem(int num)
{
    char *p1 = NULL;
    p1 = (char *)malloc(sizeof(char) * num);
    if (p1 == NULL)
    {
        return NULL;
    }
    return p1;
}

//栈

//注意 return不是把内存块 64个字节,给return出来
//而是把内存块的首地址(比如内存的编号0xaa11) ,返回给 tmp
// 理解指针的关键,是内存. 没有内存哪里来的指针

char *getMem2()
{
    char buf[64]; //临时变量 栈区存放
    strcpy(buf, "123456789");
    //printf("buf:%s\n", buf);
    return buf;
}

void main(void)
{
    char *tmp = NULL;
    tmp = getMem(10);
    if (tmp == NULL)
    {
        return ;
    }
    strcpy(tmp, "111222"); //向tmp做指向的内存空间中copy数据

    tmp = getMem2();    //tmp = 0xaa11;

    return 0;
}
```

1.3.2 全局区

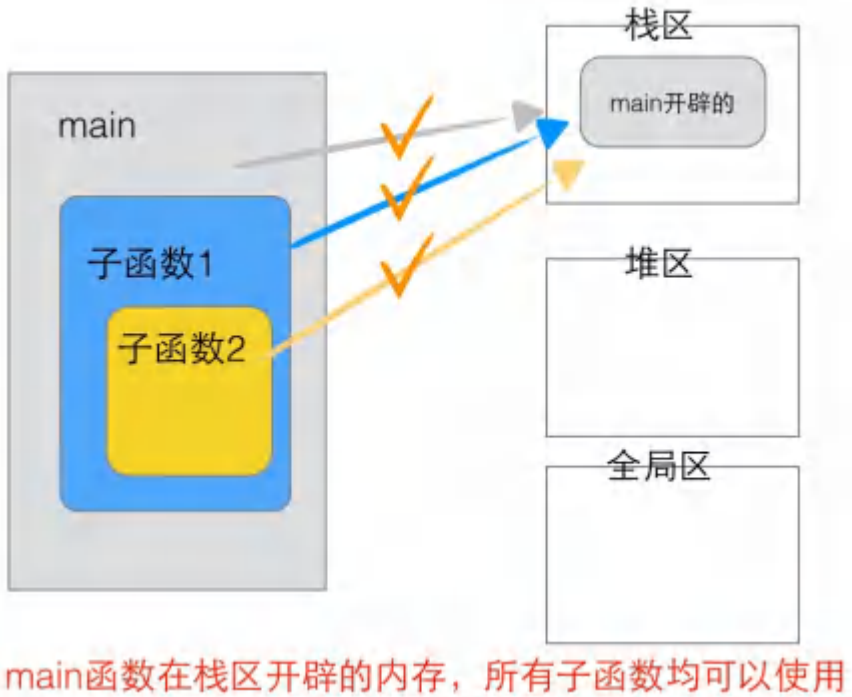
```c
#include <stdio.h>

char * getStr1()
{
    char *p1 = "abcdefg2";
    return p1;
}
char *getStr2()
{
    char *p2 = "abcdefg2";
    return p2;
}

int main(void)
{
    char *p1 = NULL;
    char *p2 = NULL;
    p1 = getStr1();
    p2 = getStr2();

    //打印p1 p2 所指向内存空间的数据
    printf("p1:%s , p2:%s \n", p1, p2);

    //打印p1 p2 的值
    printf("p1:%p , p2:%p \n", p1, p2);

    return 0;
}
```

1.4 函数的调用模型



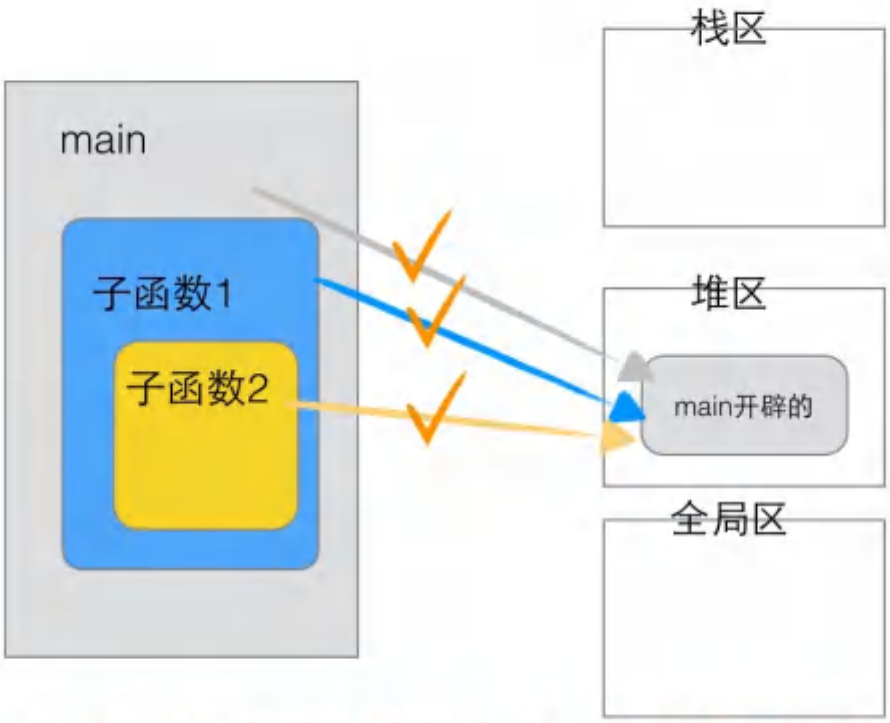
1.5 函数调用变量传递分析

（1）



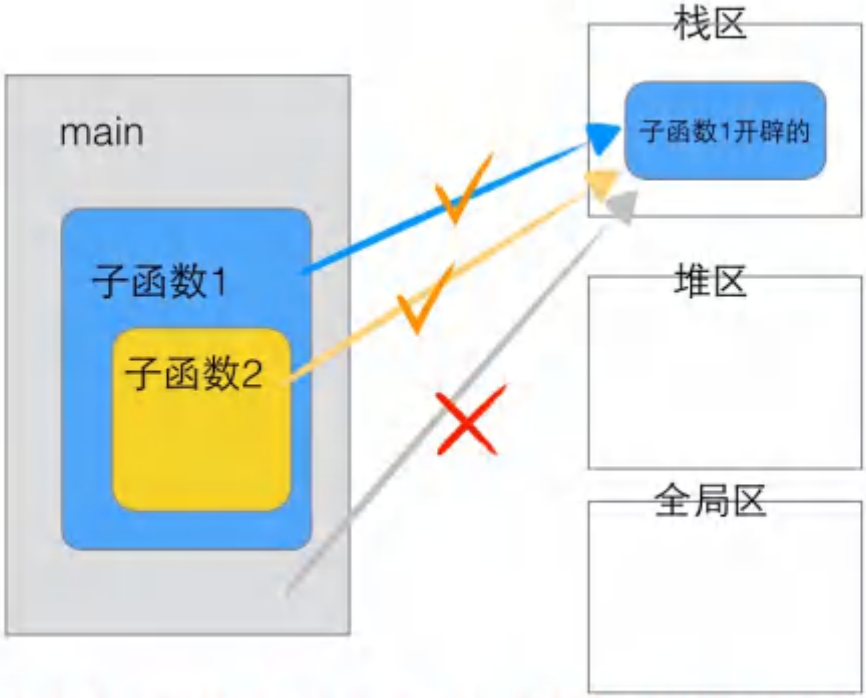
main函数在栈区开辟的内存，所有子函数均可以使用

（2）



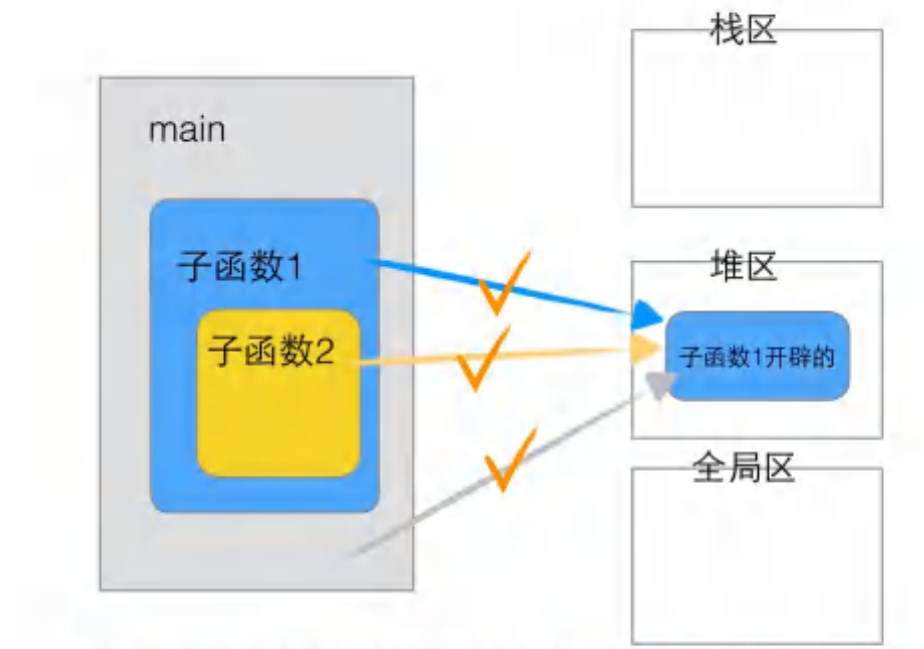
main函数在堆区开辟的内存，所有子函数均可以使用

（3）



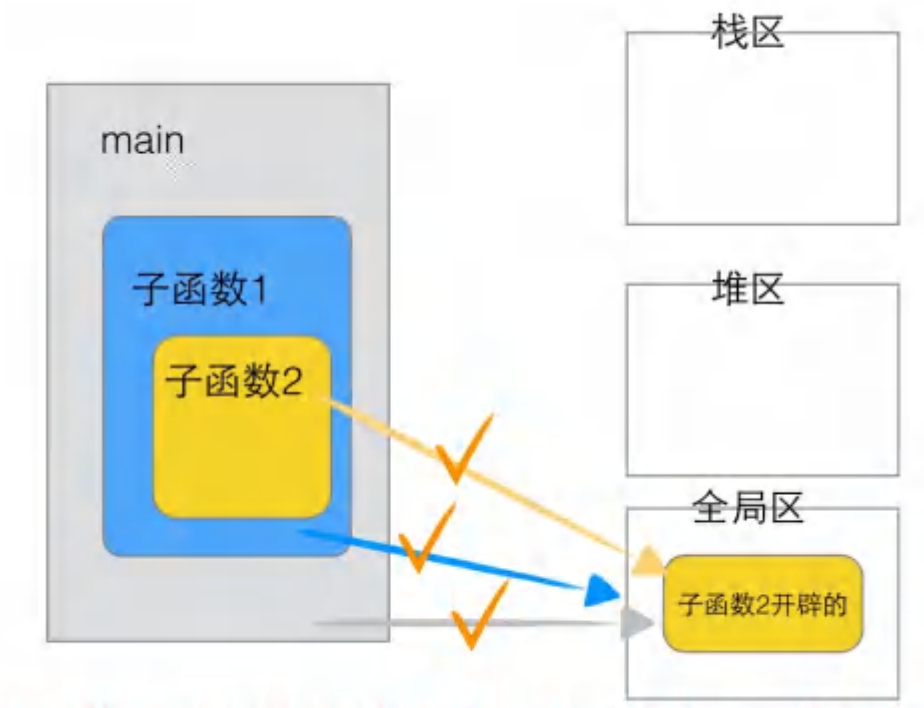
子函数1在栈区开辟的内存，子函数1和2均可以使用

（4）

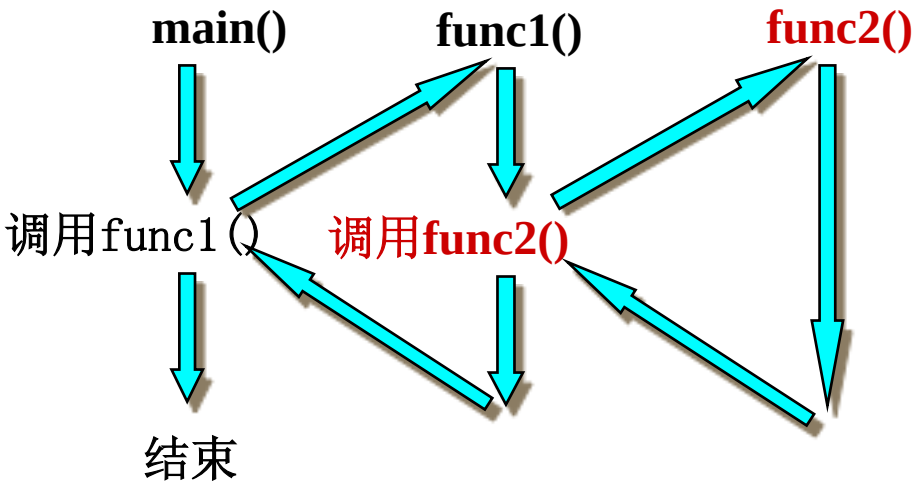


子函数1在栈区开辟的内存，子函数1和2均可以使用

（5）



子函数2在全局区开辟的内存，子函数1和main均可以使用



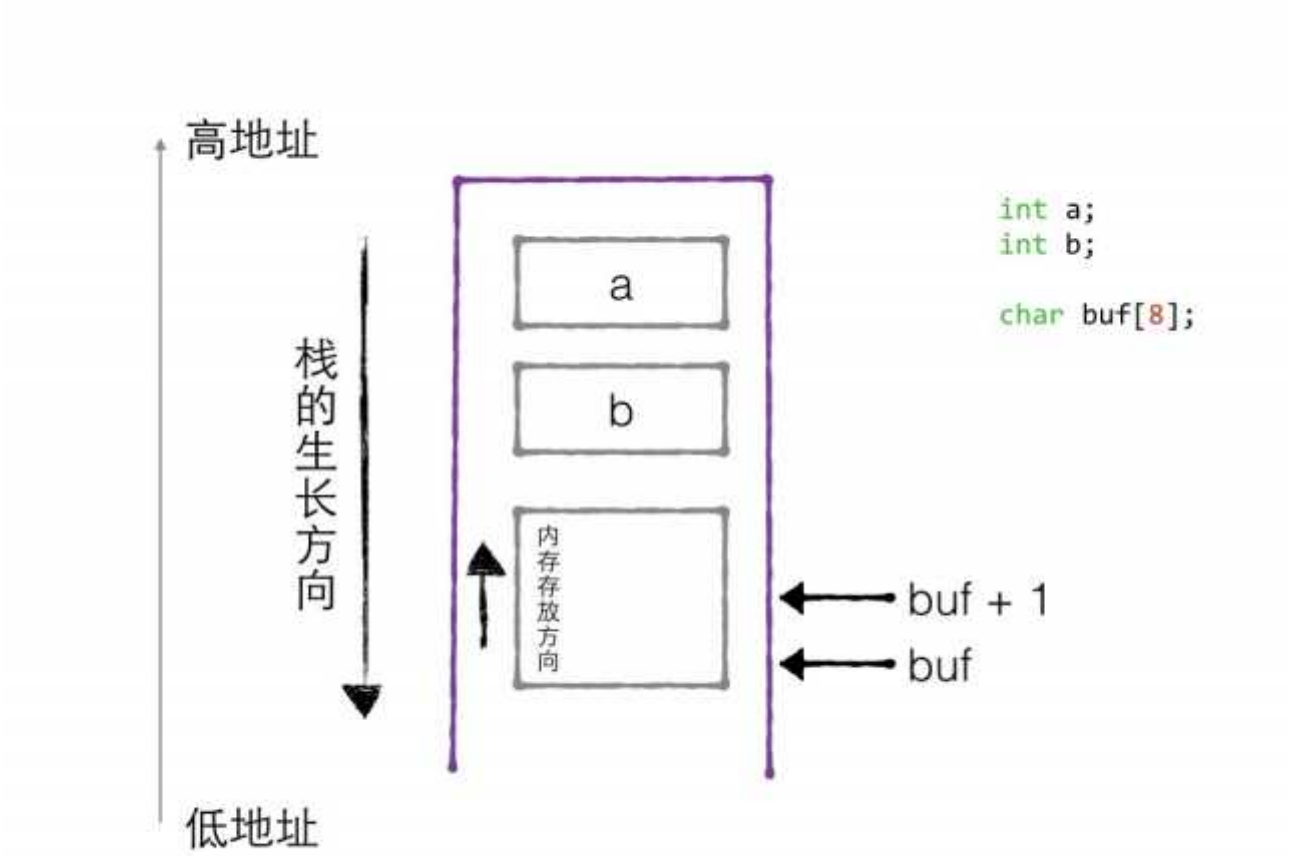
1.main函数中可以在栈/堆/全局分配内存，都可以被func1和func2使用.

2.func2在栈上分配的内存，不能被func1和main函数使用

3.func2中malloc的内存(堆),可以被main和func1函数使用。

4.func2中全局分配“abcdefg”(常量全局区)内存，可以被func1和main函数使用.

1.4 栈的生长方向和内存存放方向




```c
#include <stdio.h>

int main(void)
{
    int a;
    int b;

    char buf[4];

    printf("&a: %p\n", &a);
    printf("&b: %p\n", &b);

    printf("buf的地址 : %p\n", &buf[0]);
    printf("buf+1地址: %p \n", &buf[1]);

    return 0;
}
```

画出下面代码的内存四区图

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *get_mem(int size)
{
    char *p2 = NULL;              //分配4个字节的内存 栈区也叫临时区
    p2 = (char *)malloc(size);

    return p2;
}

int main(void)
{
    char buf[100];
    int a = 10;      //分配4个字节的内存 栈区也叫临时区
    int *p;          //分配4个字节的内存
    p = &a;          //cpu执行的代码，放在代码区

    *p = 20;

    char *mp = get_mem(100);
    strcpy(mp, "ABCDEFG");

    if (mp != NULL)
    {
        free(mp);
        mp = NULL;
    }

    return 0;
}
```

2.  夺取神器指针

2.1 指针强化

强化1：指针是一种数据类型

1） 指针也是一种变量，占有内存空间，用来保存内存地址
测试指针变量占有内存空间大小。

2）*p操作内存

> 在指针声明时，*号表示所声明的变量为指针
> 在指针使用时，*号表示 操作 指针所指向的内存空间中的值

*p相当于通过地址(p变量的值)找到一块内存，然后操作内存
*p放在等号的左边赋值（给内存赋值）
*p放在等号的右边取值（从内存获取值）

3）指针变量和它指向的内存块是两个不同的概念。

> 规则1： 给p赋值p=0x1111; 只会改变指针变量值，不会改变所指的内容；
> p = p +1; //p++
> 规则2： 给*p赋值 *p='a'; 不会改变指针变量的值，只会改变所指的内存块的值
> 规则3： =左边*p 表示 给内存赋值， =右边*p 表示取值 含义不同切结！
> 规则4： 保证所指的内存块能修改

4）指针是一种数据类型，是指它指向的内存空间的数据类型 。

```
int a;
int *p = &a;
p++;
```

指针步长（p++），根据所致内存空间的数据类型来确定.

p++ 等价于 (unsigned char)p+sizeof(a);

指针指向谁，就把谁的地址赋值给指针。

5）当我们不断的给指针变量赋值，就是不断的改变指针变量（和所指向内存空间没有任何关系）。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>


//不断给指针赋值就是不断改变指针的指向
int main(void)
{
    char    buf[128];
    int     i;

    char    *p2 = NULL;
    char    *p1 = NULL;

    p1 = &buf[0]; //不断的修改p1的值 相当于 不断改变指针的指向
    p1 = &buf[1];
    p1 = &buf[2];

    for (i=0; i<10; i++)
    {
        //不断改变p1指向的内存块
        *p1 = buf[i];
    }

    for (i=0; i<10; i++)
    {
        //不断改变p1本身变量
        p1 = &buf[i];
    }

    p2 = (char *)malloc(100);
    strcpy(p2, "abcdefg1212333333333311");

    for (i=0; i<10; i++)
    {
        //不断的改变p1本身变量，跟p1指向的内存块无关
```

```
        p1 = p2+i;
        printf("%c ", *p1);
    }

    return 0;
}
```

6）不允许向NULL和未知非法地址拷贝内存。

强化2：间接赋值（*p）是指针存在的最大意义

*p间接赋值成立条件：三大条件
条件一： 2个变量（通常一个实参，一个形参）
条件二： 建立关系，实参取地址赋给形参指针
条件三：*p形参去间接修改实参的值

```
int num = 0;
int *p = NULL;  // 条件一：两个变量

p = &num;       // 条件二：建立关系

Num = 1;

*p = 2 ;        // 条件三：通过* 操作符， 间接的给变量内存赋值
```

- 间接操作：从0级指针到1级指针

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//使用一级指针
void  getFileLen(int *p)
{
    *p = 41;    //  p的值是file_len的地址 *p的地址间接修改file_len的值
                //在被调用函数里面 通过形参 去 间接的修改 实参的值...
}

int  getFileLen2()
```

```
{
    int len = 100;
    return len;
}

//不使用指针， 0级指针
void  getFileLen3(int file_len)
{
    file_len = 100;
}

//1级指针的技术推演
int main(void)
{
    int file_len = 10;       //条件1  定义了两个变量(实参 另外一个变量是形参p)
    int *p = NULL;

    p = &file_len;           //条件2 建立关联

    file_len = 20;          //直接修改
    *p = 30;                //条件3 p的值是file_len的地址
                            //      *就像一把钥匙 通过地址
                            //      找到一块内存空间 间接的修改了file_len的值

    {
        *p = 40;  //  p的值是a的地址 *a的地址间接修改a的值  //条件3 *p
        printf("file_len: %d\n", file_len);
    }

    getFileLen(&file_len); //建立关联: 把实参取地址 传递给 形参
    printf("getFileLen后 file_len: %d \n", file_len);
    getFileLen3(file_len);
    printf("getFileLen3后 file_len: %d \n", file_len);

    return 0;
}
```

- 间接操作：从1级别指针到2级指针

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>


void getMem(char *p2)
{
    p2 = 0x80088008;
}
```

```c
void getMem2(char **p2)
{
    *p2 = 0x40044004; //间接赋值  p2是p1的地址
}

int main(void)
{
    char *p1 = NULL;
    char **p2 = NULL;

    //直接修改p1的值
    p1 = 0x11001100;

    //间接修改p1的值
    p2 = &p1;

    *p2 = 0x10101010;    //间接赋值  p2是p1的地址

    printf("p1:%p \n", p1);

    {
        *p2 = 0x20022002; //间接赋值  p2是p1的地址
        printf("p1:%p \n", p1);
    }

    getMem(p1);

    getMem2(&p1);

    printf("p1:%p \n", p1);

    return 0;
}
```

函数调用时，形参传给实参，用实参取地址，传给形参，在被调用函数里面用*p，来改变实参，把运算结果传出来。

✵ 间接赋值的推论

用1级指针形参，去间接修改了0级指针(实参)的值。
用2级指针形参，去间接修改了1级指针(实参)的值。
用3级指针形参，去间接修改了2级指针(实参)的值。
用n级指针形参，去间接修改了n-1级指针(实参)的值。

- 间接操作：应用场景

正常：　　条件一，条件二，条件三都写在一个函数里。
间接赋值：条件一，条件二写在一个函数里，　条件三写在另一个函数里

```c
#include <stdio.h>
#include <string.h>

/* 间接赋值成立的三个条件
   条件1     定义1个变量（实参）
   条件2     建立关联：把实参取地址传给形参
   条件3:    *形参去间接地的修改了实参的值。
 */


void copy_str(char *p1, char *p2)
{
    while (*p1 != '\0')
    {
        *p2 = *p1;
        p2++;
        p1++;
    }
}

//间接赋值的应用场景
int main(void)
{
    //1写在一个函数中 2作为形参建立关联    3 单独写在另外一个函数里面

    char from[128];
    char to[128] = {0};

    strcpy(from, "1122233133332fafdsafas");

    copy_str(from, to);

    printf("to:%s \n", to);

    return 0;
}
```

强化3：理解指针必须和内存四区概念相结合

1）　主调函数 被调函数
　　a)　　主调函数可把堆区、栈区、全局数据内存地址传给被调用函数
　　b)　　被调用函数只能返回堆区、全局数据

2）　内存分配方式

a)　指针做函数参数，是有输入和输出特性的。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int getMem(char **myp1, int *mylen1,  char **myp2, int *mylen2)
{
    int     ret = 0;
    char    *tmp1, *tmp2;

    tmp1 = (char *)malloc(100);
    strcpy(tmp1, "1132233");

    //间接赋值
    *mylen1 = strlen(tmp1);  //1级指针
    *myp1 = tmp1;            //2级指针的间接赋值

    tmp2 = (char *)malloc(200);
    strcpy(tmp2, "aaaaavbddddddddd");

    *mylen2 = strlen(tmp2);  //1级指针
    *myp2 = tmp2;            //2级指针的间接赋值

    return ret;
}


int  main(void)
{
    int     ret = 0;
    char    *p1 = NULL;
    int     len1 = 0;
    char    *p2 = NULL;
    int     len2 = 0;

    ret = getMem(&p1, &len1, &p2, &len2);
    if (ret != 0)
    {
        printf("func getMem() err:%d \n", ret);
        return ret;
    }

    printf("p1:%s \n", p1);
    printf("p2:%s \n", p2);

    if (p1 != NULL)
    {
        free(p1);
        p1 = NULL;
    }
    if (p2 != NULL)
    {
```

```
        free(p2);
        p2 = NULL;
    }


    return 0;
}
```

强化4：应用指针必须和函数调用相结合
（指针做函数参数）

指针作为函数参数是研究指针的重点。

如果指针是子弹，那么函数就是枪管，子弹只有在枪管中才能发挥出威力。

一级指针典型用法：

一级指针做输入

```
int showbuf(char *p);
int showArray(int *array, int iNum);
```

一级指针做输出

```
int getLen(char *pFileName, int *pfileLen);
```

理解

输入：主调函数分配内存

输出：被调用函数分配内存

被调用函数是在heap上分配内存而非stack上

二级指针典型用法：

二级指针做输入

```
int main(int arc ,char *arg[]);          //字符串数组
int shouMatrix(int [3][4], int iLine);
```

二级指针做输出

```
int Demo64_GetTeacher(Teacher **ppTeacher);
int Demo65_GetTeacher_Free(Teacher **ppTeacher);
int getData(char **data, int *dataLen);
int getData_Free(void *data);
int getData_Free2(void **data);              //避免野指针
```

	内存分配方式	主调函数 实参	被调函数 形参	备注
1级指针 （输入）	堆	分配	使用	一般应用禁止
	栈	分配	使用	常用
	`int showbuf(char *p);` `int showArray(int *array, int iNum);`			
1级指针 （输出）	栈	使用	结果传出	常用
	`int getLen(char *pFileName, int *pfileLen);`			
2级指针 （输入）	堆	分配	使用	一般应用禁用
	栈	分配	使用	常用
	`int main(int arc ,char *arg[]);` //指针数组 `int shouMatrix(int [3][4], int iLine);` //二维字符串数组			
2级指针 （输出）	堆	使用	分配	不建议使用，转化为一级指针输出

	`int getData(char **data, int *dataLen);`
	`int getData_Free(void *data);`
	`int getData_Free(void **data);` //避免野指针			

3级指针 （输出）	堆	使用	分配	不常用
	`int getFileAllLine(char ***content, int *pLine);` `int getFileAllLine_Free(char ***content, int *pLine);`			

2.2 经典语录

1）指针也是一种数据类型，指针的数据类型是指它所指向内存空间的数据类型

2）间接赋值*p是指针存在的最大意义

3）理解指针必须和内存四区概念相结合

4）应用指针必须和函数调用相结合（指针做函数参数）

指针是子弹，函数是枪管；子弹只有沿着枪管发射才能显示它的威力；指针的学习重点不言而喻了吧。接口的封装和设计、模块的划分、解决实际应用问题；它是你的工具。

5）指针指向谁就把谁的地址赋给指针

6）C/C++语言有它自己的学习特点；若java语言的学习特点是学习、应用、上项目；那么C/C++语言的学习特点是：学习、理解、应用、上项目。多了一个步骤。

7）理解指针关键在内存，没有内存哪来的内存首地址，没有内存首地址，哪来的指针。

3.  收纳小鬼字符串

3.1 字符串的基本操作

3.1.1 字符数组初始化方法

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//一级指针的典型用法
//字符串
//1 C语言的字符串 以零'\0'结尾的字符串
//2 在C语言中没有字符串类型  通过字符数组 来模拟字符串
//3 字符串的内存分配  堆上 栈上 全局区


//字符数组 初始化
int main()
{
    //1 不指定长度  C编译器会自动帮程序员 求元素的个数
    char buf1[] = {'a', 'b', 'c', 'd'};  //buf1是一个数组 不是一个以0结尾的字符串

    //2 指定长度
    char buf2[100] = {'a', 'b', 'c', 'd'};
    //后面的buf2[4]-buf2[99] 个字节均默认填充0

    //char buf3[2] = {'a', 'b', 'c', 'd'};
    //如果初始化的个数大于内存的个数 编译错误

    printf("buf1: %s\n", buf1);
    printf("buf2: %s \n", buf2);
    printf("buf2[88]:%d \n", buf2[88]);


    //3 用字符串初始化 字符数组
    char buf3[] = "abcd";        //buf3 作为字符数组 有5个字节
                                 //       作为字符串有 4个字节

    int len = strlen(buf3);
    printf("buf3字符的长度:%d \n", len);     //4

    //buf3 作为数组 数组是一种数据类型 本质(固定小大内存块的别名)
    int size = sizeof(buf3); //
    printf("buf3数组所占内存空间大小:%d \n", size); //5
```

```
    char buf4[128] = "abcd"; // buf
    printf("buf4[100]:%d \n", buf4[100]);

    return 0;
}
```

sizeof 和 strlen的区别

1. sizeof为一个操作符，执行sizeof的结果，在编译期间就已经确定；
   strlen是一个函数，是在程序执行的时候才确定结果。

2. sizeof和strlen对于求字符串来讲，
   sizeof() 字符串类型的大小，包括’\0’ ；
   strlen() 字符串的长度不包括’\0’ 。

3.1.2 数组法和指针法操作字符串

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int     i = 0;
    char    *p = NULL;
    char    buf[128] = "abcdefg";

    //通过[]
    for (i=0; i<strlen(buf); i++)
    {
        printf("%c ", buf[i]);
    }

    p = buf; //buf 代表数组首元素的地址
    //通过指针
    for (i=0; i<strlen(buf); i++)
    {
        p = p +i;
        printf("%c ", *p ) ;
    }

    //通过数组首元素地址 buf 来操作
    for (i=0; i<strlen(buf); i++)
    {
```

```
        printf("%c ", *(buf+i) ) ;
    }

    return 0;
}
```

- []的本质 和*p 是一样

- buf 是一个指针, 只读的常量。之所以buf是一个常量指针是为了释放内存的时候,保证buf所指向的内存空间安全

练习题:
画出字符串一级指针内存四区模型

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char buf[20]= "aaaa";
    char buf2[] = "bbbb";
    char *p1 = "111111";
    char *p2 = malloc(100);

    strcpy(p2, "3333");

    return 0;
}
```

3.3 字符串做函数参数

我们来看看简单的copy一条字符串实现。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    char a[] = "i am a student";
    char b[64];
    int  i = 0;

    for (i=0; *(a+i) != '\0'; i++)
    {
        *(b+i) = *(a+i);
    }

    //0没有copy到b的buf中.

    b[i] = '\0'; //重要

    printf("a:%s \n", a);
    printf("b:%s \n", b);

    return 0;
}
```

以上是拷贝一个字符串用到了数组名（也就是数组首元素地址）的偏移来完成。

那么如果用一个一级指针呢？

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//字符串copy函数

//form形参 形参to 的值 不停的在变化....
//不断的修改了from和to的指向
void copy_str1(char *from, char *to)
{
    for (; *from!='\0'; from++, to++)
    {
        *to = *from;
    }
```

```
    *to = '\0';

    return ;
}

void copy_str2(char *from, char *to)
{
    for (; *from!='\0';)
    {
        *to++ = *from++;  //  先 *to = *from;  再from++, to++
    }
    *to = '\0'; //

    return ;
}

void copy_str3(char *from, char *to)
{
    while( (*to = *from) != '\0' )
    {
        from ++;
        to ++;
    }
}

void copy_str4(char *from , char *to)
{
    while ( (*to++ = *from++) != '\0')
    {
        ;
    }
}

void copy_str5(char *from , char *to)
{
    //*(0) = 'a';
    while ( (*to++ = *from++) )
    {
        ;
    }
}


void copy_str5_err(char *from , char *to)
{
    //*(0) = 'a';
    while ( (*to++ = *from++) )
    {
        ;
    }

    printf("from:%s \n", from);

}
```

```c
//不要轻易改变形参的值，要引入一个辅助的指针变量. 把形参给接过来.....
int copy_str6(char *from , char *to)
{
    //*(0) = 'a';
    char *tmpfrom = from;
    char *tmpto = to;
    if ( from == NULL || to == NULL)
    {
        return -1;
    }


    while (*tmpto++ = *tmpfrom++) ;  //空语句

    printf("from:%s \n", from);

    return 0;
}

int main(void)
{
    int ret = 0;
    char *from = "abcd";
    char buf[100];

#if 0
    copy_str1(from, buf);
    copy_str2(from,buf);
    copy_str3(from, buf);
    copy_str4(from, buf);
    copy_str5(from ,buf);
#endif

    printf("buf:%s \n", buf);

    {
        //错误案例
        char *myto = NULL;  //要分配内存
        //copy_str25(from,myto);
    }

    {
        char *myto = NULL;  //要分配内存

        ret = copy_str6(from, myto);
        if (ret != 0)
        {
            printf("func copy_str6() err:%d  ", ret);
            return ret;
        }
    }

    return 0;
}
```

3.4 项目开发常用字符串应用模型

3.4.1 strstr中的while和do-while模型

利用strstr标准库函数找出一个字符串中substr出现的个数。

do-while模型

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    //strstr(str, str2)

    int ncount = 0;

    //初始化 让p指针达到查找的条件
    char *p = "11abcd111122abcd3333322abcd3333322qqq";

    do
    {
        p = strstr(p, "abcd");
        if (p != NULL)
        {
            ncount++; //
            p = p + strlen("abcd"); //指针达到下次查找的条件
        }
        else
        {
            break;
        }
    } while (*p != '\0');


    printf("ncount:%d \n", ncount);

    return 0;
}
```

while模型

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

```c
int main(void)
{
    int ncount = 0;

    //初始化 让p指针达到查找的条件
    char *p = "2abcd3333322qqqabcd";

    while ( (p = strstr(p, "abcd")) != NULL )
    {
        ncount ++;
        p = p + strlen("abcd"); //让p指针达到查找的条件
        if (*p == '\0')
        {
            break;
        }
    }
    printf("ncount:%d \n", ncount);

    return 0;
}
```

练习：

求字符串p中 abcd出现的次数

1 请自定义函数接口，完成上述需求

2 自定义的业务函数 和 main函数必须分开

3.4.2 两头堵模型

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int get_count_non_space(char *str, int * count_p)
{
    int i = 0;
    int j = strlen(str)-1;
    int ncount = 0;

    if (str == NULL || count_p == NULL) {
        fprintf(stderr, "str == NULL, count_p == NULL\n");
        return -1;
    }

    while (isspace(str[i]) && str[i] != '\0')
    {
        i++;
    }

    while (isspace(str[j]) && j > i)
    {
        j--;
    }

    ncount = j-i+1;

    *count_p = ncount;

    return 0;
}

int main(void)
{
    char *p = "      abcdefg      ";

    int ret = 0;
    int ncount = 0;

    ret = get_count_non_space(p, &ncount);
    if (ret < 0) {
        fprintf(stderr, "get_count_non_space err, ret = %d\n", ret);
        return 0;
    }

    printf("ncount = %d\n", ncount);

    return 0;
}
```

练习：

有一个字符串开头或结尾含有n个空格

（” abcdefgdddd ”），

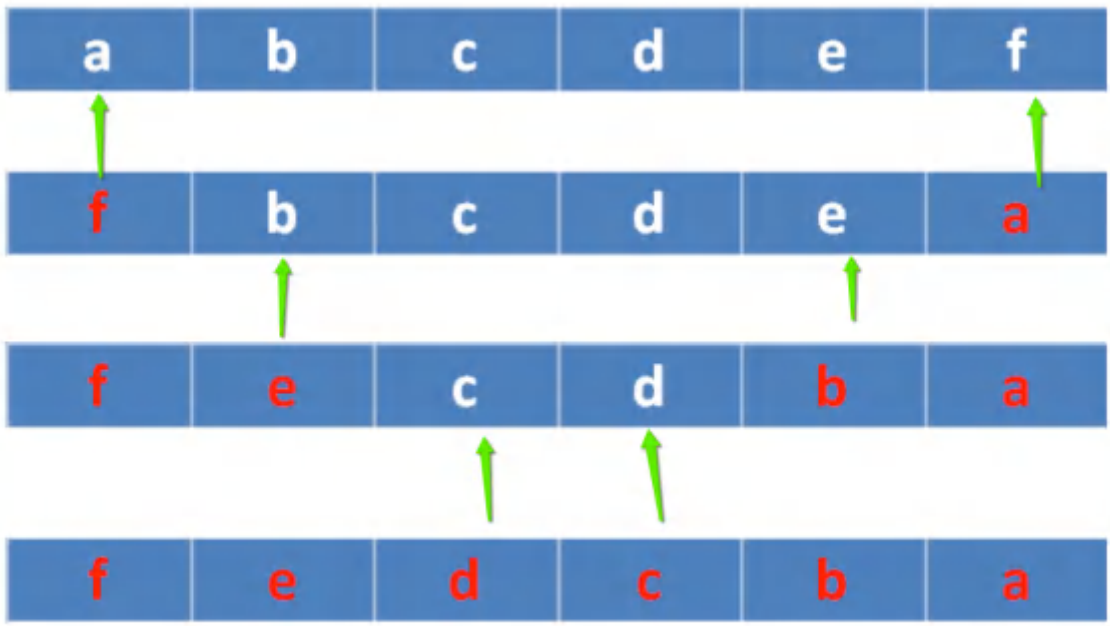
欲去掉前后空格，返回一个新字符串。

要求1：请自己定义一个接口（函数），并实现功能；

要求2：编写测试用例。

int trimSpace(char *inbuf, char *outbuf);

3.4.3 字符串反转模型




```
char* inverse(char *str)
{
    int length = strlen(str);
    char *p1 = NULL;
    char *p2 = NULL;
    char tmp_ch;

    if (str == NULL) {
        return NULL;
    }

    p1 = str;
    p2 = str + length -1;
```

```
    while (p1 < p2) {
        tmp_ch = *p1;
        *p1 = *p2;
        *p2 = tmp_ch;
        ++p1;
        --p2;
    }

    return str;
}
```

递归思想

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char g_buf[1024];

//3 递归和非全局变量
int inverse4(char *str, char *dst)
{
    if (str == NULL) {
        return -1;
    }

    if (*str == '\0') {// 递归技术的条件
        return 0;
    }

    if (inverse4(str+1, dst)  < 0 ) {
        return -1;
    }

    strncat(dst, str, 1);
```

```
    return 0;
}

//2 递归和全局变量(把逆序的结果放在全局变量里)
int inverse3(char *str)
{
    if (str == NULL) {
        return -1;
    }

    if (*str == '\0') {// 递归技术的条件
        return 0;
    }

    if (inverse3(str+1)  < 0 ) {
        return -1;
    }

    strncat(g_buf, str, 1);

    return 0;
}

//1 通过递归的方式逆序打印
int inverse2(char *str)
{
    if (str == NULL) {
        return -1;
    }

    if (*str == '\0') {// 递归技术的条件
        return 0;
    }

    if (inverse2(str+1)  < 0 ) {
        return -1;
    }
    printf("%c", *str);
    return 0;
}

int main(void)
{
    char buf[] = "abcdefg";
    char dst_buf[128] = {0};

    // test inverse()
    //printf("buf : %s\n", inverse(buf));

    // test inverse2();
    //inverse2(buf);

    // test inverse3();
    //inverse3(buf);
    //printf("g_buf : %s\n", g_buf);
```

```
    // test inverse4();
    inverse4(buf, dst_buf);
    printf("dst_buf : %s\n", dst_buf);


    return 0;
}
```

3.5 一级指针(char*)易错地方

- 对空字符串和非法字符串的判断

```
#include <stdio.h>

void copy_str(char *from, char *to)
{
    if (*from == '\0' || *to == '\0')
    {
        printf("func copy_str() err\n");
        return;
    }

    for (; *from!='\0'; from++, to++)
    {
        *to = *from;
    }
    *to = '\0';
}

int main()
{
    char *p ="aabbccdd";
    char to[100];
    copy_str(p, to);

    printf("to : %s\n", to);

    return 0;
}
```

* 越界

```
char buf[3] = "abc";
```

* 指针的叠加会不断改变指针的方向

```
char *getKeyByValue(char **keyvaluebuf, char *keybuf)
{
    int i = 0;
    char *a = (char *)malloc(50);

    for (; **keyvaluebuf != '\0'; i++)
    {
        *a++ = *(*keyvaluebuf)++;
    }

    free(a);
}
```

```
void copy_str_err(char *from, char *to)
{
    for (; *from!='\0'; from++, to++)
    {
        *to = *from;
    }
    *to = '\0';
    printf("to:%s", to);
    printf("from:%s", from);
}
```

* 局部变量不要外传

```
char *my_stract(char *x, char* y)
{
    char str[80];
    char *z=str;            /*指针z指向数组str*/

    while(*z++=*x++);
    z--;                     /*去掉串尾结束标志*/
```

```
    while(*z++=*y++);
    z=str;                  /*将str地址赋给指针变量z*/

    return(z);
}
```

* 函数内使用辅助变量的重要性

```
#include <stdio.h>
#include <string.h>

//char *p = "abcd11111abcd2222abcdqqqqq";
//字符串中"abcd"出现的次数。
int getSubCount(char *str, char *substr, int *mycount)
{
    int ret = 0;
    char *p = str;
    char *sub = substr;

    if (str==NULL || substr==NULL || mycount == NULL)
    {
        ret = -1;
        return ret;
    }

    do
    {
        p = strstr(p, sub);
        if (p!= NULL)
        {
            *mycount++;
            p = p + strlen(sub);
        }
        else
        {
            break;
        }
    } while (*p != '\0');

    return ret;
}

int main(void)
{
    char *p = "abcd11111abcd2222abcdqqqqq";
    int count = 0;

    if (getSubCount(p, "abcd", &count) < 0) {
```

```
        printf("error\n");
    }

    printf("abcd's count :%d\n", count);

    return 0;
}
```

3.6 谈谈const

* const 知识点

一. const声明的变量只能被读

```
const int i=5;
int j=0;

i=j;   //非法，导致编译错误
j=i;   //合法
```

二．必须初始化

```
const int i=5;    //合法
const int j;      //非法，导致编译错误
```

三．如何在另一.c源文件中引用const常量

```
extern const int i;         //合法
extern const int j=10;      //非法，常量不可以被再次赋值
```

四．可以避免不必要的内存分配

```
#define STRING "abcdefghijklmn"
const char string[]="ABCDEFGHIJK";

printf(STRING);   //为STRING分配了第一次内存
printf(string);   //为string一次分配了内存，以后不再分配

printf(STRING);   //为STRING分配了第二次内存
printf(string);
```

```
/*
由于const定义常量从汇编的角度来看，只是给出了对应的内存地址，
而不是象#define一样给出的是立即数，所以，const定义的常量在
程序运行过程中只有一份拷贝，而#define定义的常量在内存中有
若干个拷贝。
*/
```

五．C语言中const是一个冒牌货

通过强制类型转换，将地址赋给变量，再作修改即可以改变const常量值。

* const变量一览

```
int main(void)
{
    const int a;     //代表一个常整型数
    int const b;     //代表一个常整型数

    const char *c;   // 是一个指向常整型数的指针
                     // (所指向的内存数据不能修改，
                     //  但是本身可以修改)

    char * const d; // 常指针(指针变量不能被修改，
                     //         但是它所指向的内存空间可以被修改)

    const char * const e;// 一个指向常量整型的常指针
                         // (指针和它所指向的内存空间，
                         //  均不可以修改)

    return 0;
}
```

* const好处

合理的利用const，
1 指针做函数参数，可以有效的提高代码可读性，减少bug；
2 清楚的分清参数的输入和输出特性

3.7 强化练习

1、有一个字符串“1a2b3d4z”；

要求写一个函数实现如下功能：

功能1：把偶数位字符挑选出来，组成一个字符串1。
功能2：把奇数位字符挑选出来，组成一个字符串2。
功能3：把字符串1和字符串2，通过函数参数，传送给main，并打印。
功能4：主函数能测试通过。

```
int getStr1Str2(char *source, char *buf1, char *buf2);
```

2、键值对（“key = value”）字符串，在开发中经常使用.

要求1：请自己定义一个接口，实现根据key获取.
要求2：编写测试用例。
要求3：键值对中间可能有n多空格，请去除空格

注意：键值对字符串格式可能如下：

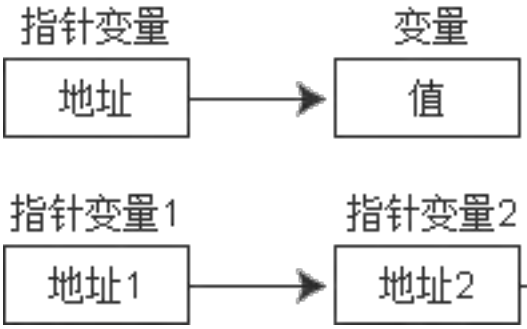
```
"key1 = value1"
"key2 =       value2"
"key3  = value3"
"key4        = value4"
"key5   =   "
"key6   ="

int getKeyByValue(char *keyvaluebuf,  char *keybuf,  char *valuebuf, int *
valuebuflen);

int main(void)
{
    //...
    getKeyByValude("key1 = valude1", "key1", buf, &len);
    //...

    return 0;
}
```

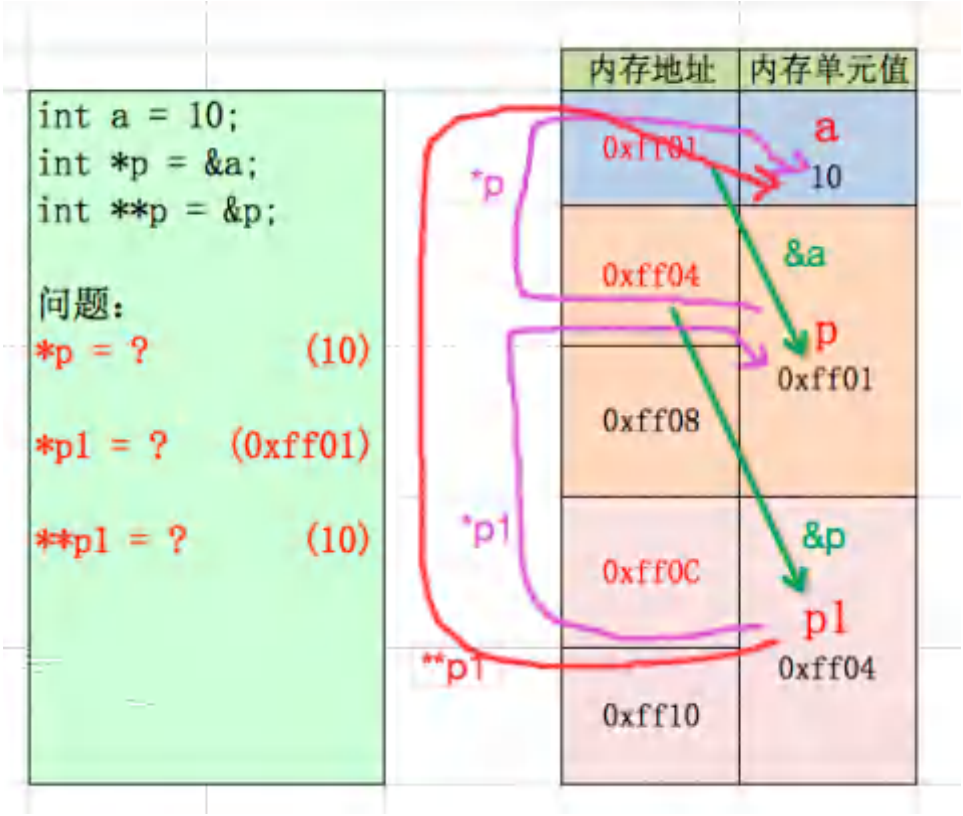
4.  打造神器二级指针

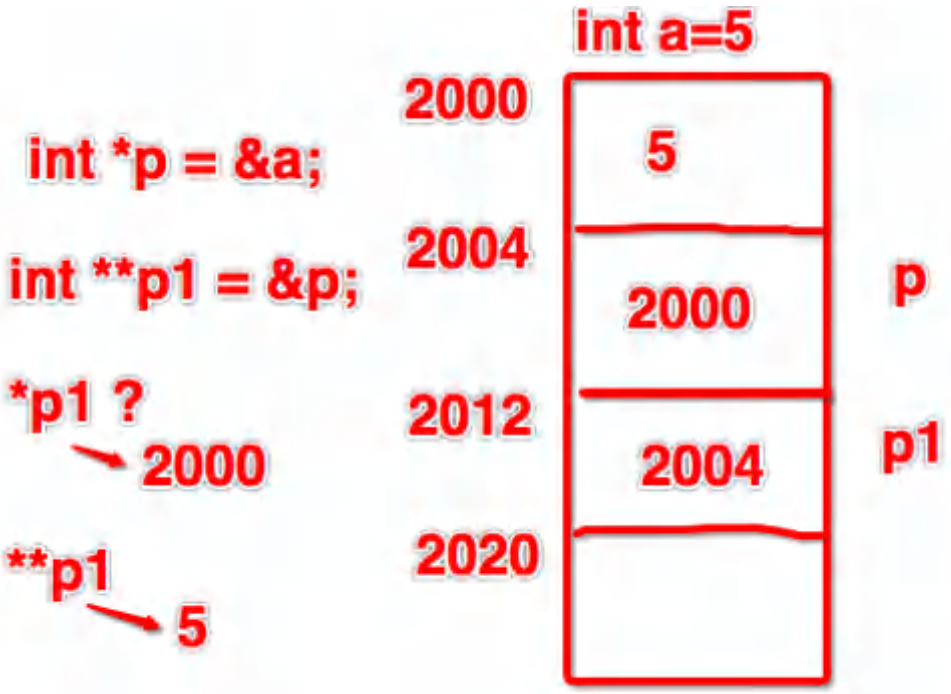
4.1 二级指针基本概念

如果一个指针变量存放的又是另一个指针变量的地址,则称这个指针变量为指向指针的指针 变量。也称为“二级指针”。

通过指针访问变量称为间接访问。由于指针变量直接指向变量,所以称为“一级指针”。而 如果通过指向指针的指针变量来访问变量则构成“二级指针”。




int a = 10;
int *p = &a;
int **p = &p;
问题：
*p = ?　　(10)
*p1 = ?　　(0xff01)
**p1 = ?　　(10)
内存地址
内存单元值
0xff01
a
10
0xff04
&a
p
0xff01
0xff08
0xff0C
&p
p1
0xff04
0xff10
*p
*p1
**p1


int a=5
2000
5
2004
2000
p
2012
2004
p1
2020
int *p = &a;
int **p1 = &p;
*p1 ?
2000
**p1
5

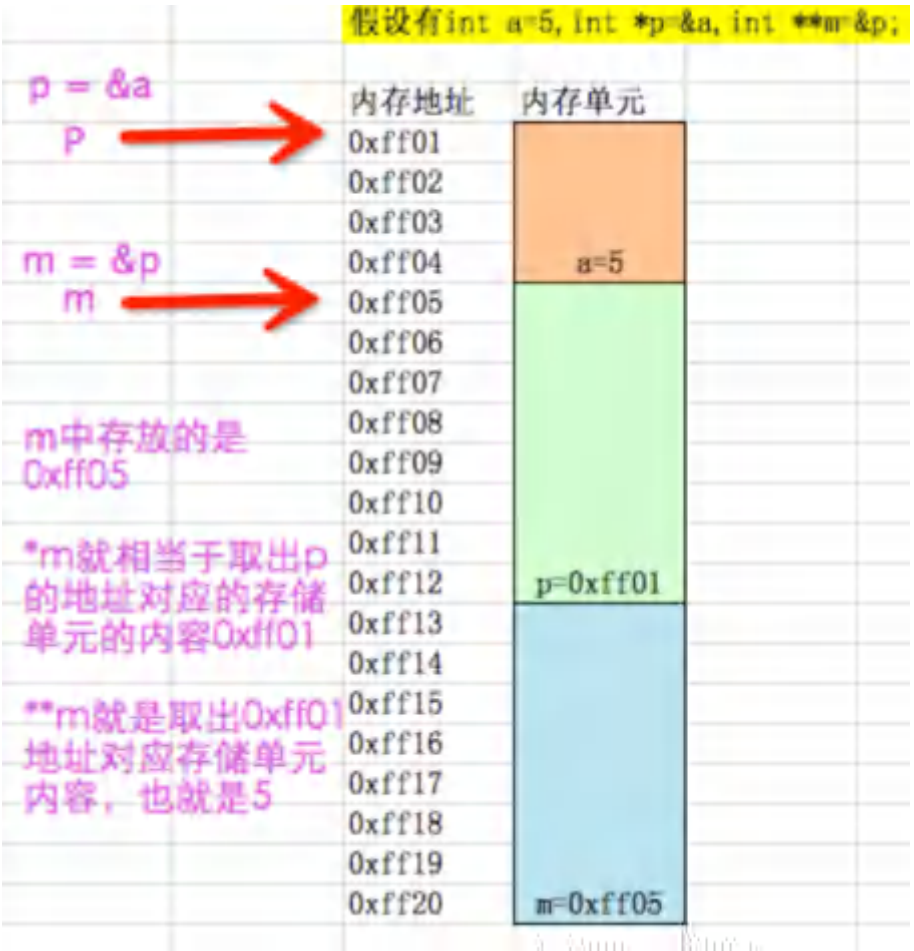
```c
int main(int argc, char * argv[])
{
    int a=5;
    int* p=&a;
    int **m=&p; //*(*m)
    printf("&a = %p\n",&a);
    printf("p = %p\n",p);
    printf("&p = %p\n",&p);
    printf("m = %p\n",m);
    printf("m = %p\n",m);
    printf("*m = %p\n",*m);
    printf("%d\n",**m);

    return 0;
}
```

```
&a = 0x7fff54d60bcc
p = 0x7fff54d60bcc
&p = 0x7fff54d60bc0
m = 0x7fff54d60bc0
m = 0x7fff54d60bc0
*m = 0x7fff54d60bcc
**m = 5
```

m=p的地址，m=&p

*m=p指向的内存空间内容，就是&a

**m=a变量的值



4.2 二级指针输出特性

```
int getMem(/*out*/char **myp1, /*out*/int *mylen1,
           /*out*/char **myp2, /*out*/int *mylen2);

int getMem_Free(char **myp1);
```

4.3 二级指针输入特性

第一种输入模型

```
char *myArray[] = {"aaaaaa", "ccccc", "bbbbbb", "111111"};

void printMyArray(char **myArray, int num);
void sortMyArray(char **myArray, int num);
```

第二种输入模型

```
char myArray[10][30] = {"aaaaaa", "ccccc", "bbbbbbb", "1111111111111"};

void printMyArray(char myArray[10][30], int num);
void sortMyArray(char myArray[10][30], int num);
```

第三种输入模型

```
char **myArray = NULL;

char **getMem(int num);
void printMyArray(char **myArray, int num);
void sortMyArray(char **myArray, int num);
void arrayFree(char **myArray, int num);
```

练习：通过下列代码画出三种二级指针模型的内存四区图

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    int i = 0;

    //指针数组
    char *p1[] = {"123", "456", "789"};

    //二维数组
    char p2[3][4]  = {"123", "456", "789"};

    //手工二维内存
    char **p3 = (char **)malloc(3 * sizeof(char *)); //int array[3];

    for (i=0; i<3; i++)
    {
        p3[i] = (char *)malloc(10*sizeof(char)); //char buf[10]

        sprintf(p3[i], "%d%d%d", i, i, i);
    }

    return 0;
}
```

4.3 多级指针

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int getMem(char ***p , int num)
{
    int i = 0;
    char **tmp = NULL;

    if (p == NULL)
    {
        return -1;
    }

    tmp = (char **)malloc(sizeof(char *) * num);
    if (tmp == NULL)
    {
        return -1;
    }

    for (i=0; i<num; i++)
    {
        tmp[i] = (char *)malloc(sizeof(char)  * 100  );
        sprintf(tmp[i], "%d%d%d", i+1, i+1, i+1);
    }
    *p = tmp;

    return 0;
}

void memFree(char ***p , int num)
{
    int i = 0;
    char **tmp = NULL;

    if (p == NULL)
    {
        return ;
    }
    tmp = *p;

    for (i=0; i<num; i++)
    {
        free(tmp[i]);
    }
    free(tmp);

    *p = NULL; //把实参赋值成null
}

int main(void)
{
```

```
    int i = 0;
    char **array = NULL;
    int num = 5;

    getMem(&array, num);

    for (i=0; i<num; i++)
    {
        printf("%s \n", array[i]);
    }

    memFree(&array, num);

    return 0;
}
```

作业一：

有字符串有以下特征（“abcd11111abcd2222abcdqqqqq”），求写一个函数接口，输出以下结果。

把字符串替换成（dcba11111dcba2222dcbaqqqqq），并把结果传出。

要求：

1. 正确实现接口和功能
2. 编写测试用例

```
/*
src:     原字符串
dst:     生成的或需要填充的字符串
sub:     需要查找的子字符串
new_sub:提换的新子字符串

return : 0 成功
         -1 失败
 */
int replaceSubstr(/* in */char *src, /* out */char** dst,
                  /* in */char *sub,  /* in */char *new_sub);
```

作业二：

有一个字符串符合以下特征（“abcdef,acccd,eeee,aaaa,e3eeee,ssss,”）

写两个函数(API)，输出以下结果

第一个API

1)以逗号分隔字符串，形成二维数组，并把结果传出

2)把二维数组行数运算结果也传出

第二个API

1)以逗号分隔字符串，形成一个二级指针。

2)把一共拆分多少行字符串个数传出

要求：

1，能正确表达功能的要求，定义出接口。

2，正确实现接口和功能.

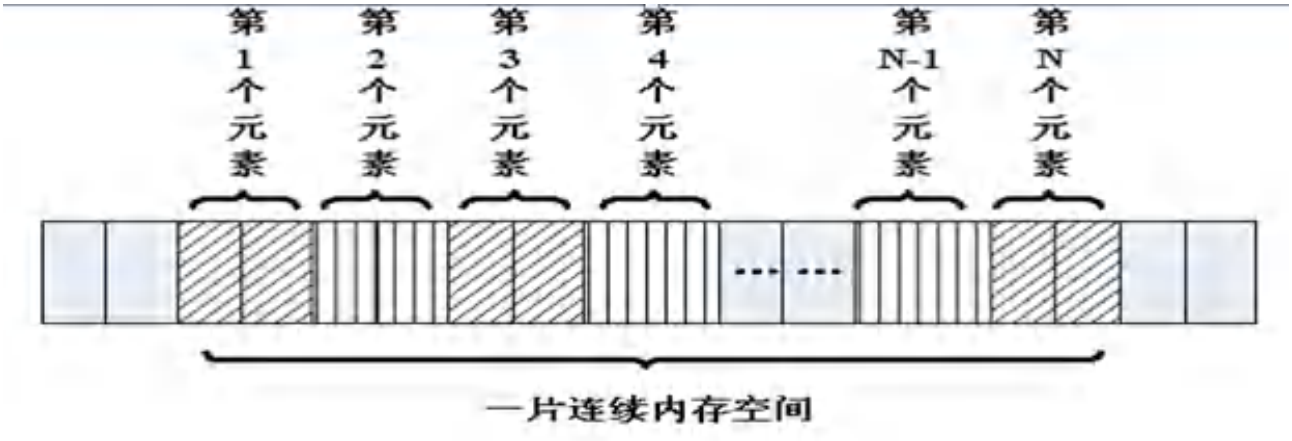
3，编写正确的测试用例.

5. 探索多维的数组

5.1 数组的基本概念

5.1.1 概念

- 元素类型角度：数组是相同类型的变量的有序集合
- 内存角度：连续的一大片内存空间



5.1.1 二维数组

游戏中的二维数组

农业中的二维数组

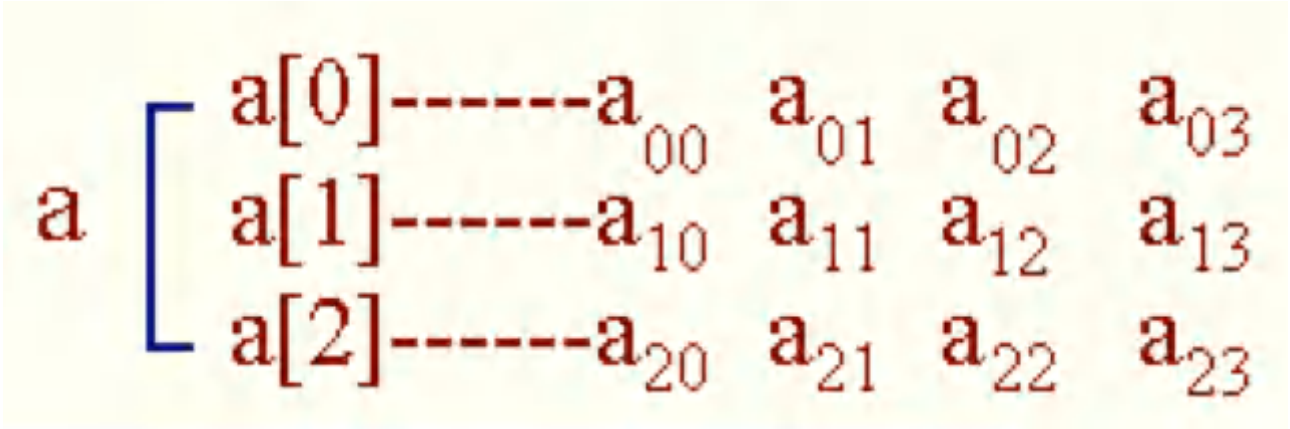
所谓多维数组就是二维和大于二维的数组,在C语言中并不直接支持多维数组,包括二维数 组。多维数组的声明是使用一维数组的嵌套声明实现的。一个一维数组的每个元素又被声明为一 维数组,从而构成二维数组,可以说二维数组是特殊的一维数组。

二维数组定义的一般形式是:

类型说明符 数组名[常量表达式1][常量表达式2]

其中常量表达式1表示第一维下标的长度,常量表达式2 表示第二维下标的长度。

例如:int a[3][4]; 说明了一个三行四列的数组,数组名为a,其下标变量的类型为整型。该数组的下标变量共有3×4 个,即:



再如人站队构成的二维数组:

	队员1	队员2	队员3	队员4	队员5	队员6
1分队	2456	1847	1243	1600	2346	2757
2分队	3045	2018	1725	2020	2458	1436
3分队	1427	1175	1046	1976	1477	2018

5.1.2 数组名

数组首元素的地址和数组地址是两个不同的概念

数组名代表数组首元素的地址，它是个常量。

变量本质是内存空间的别名，一定义数组，就分配内存，内存就固定了。所以数组名起名以后就不能被修改了。

数组首元素的地址和数组的地址值相等。

怎么样得到整个一维数组的地址？

```
int a[10];
printf("得到整个数组的地址a: %d \n", &a);
printf("数组的首元素的地址a: %d \n", a);
```

怎么样表达int a[10]这种数据类型那？

5.2 数组类型

数组的类型由元素类型和数组大小共同决定

int array[5] 的类型为 int[5];

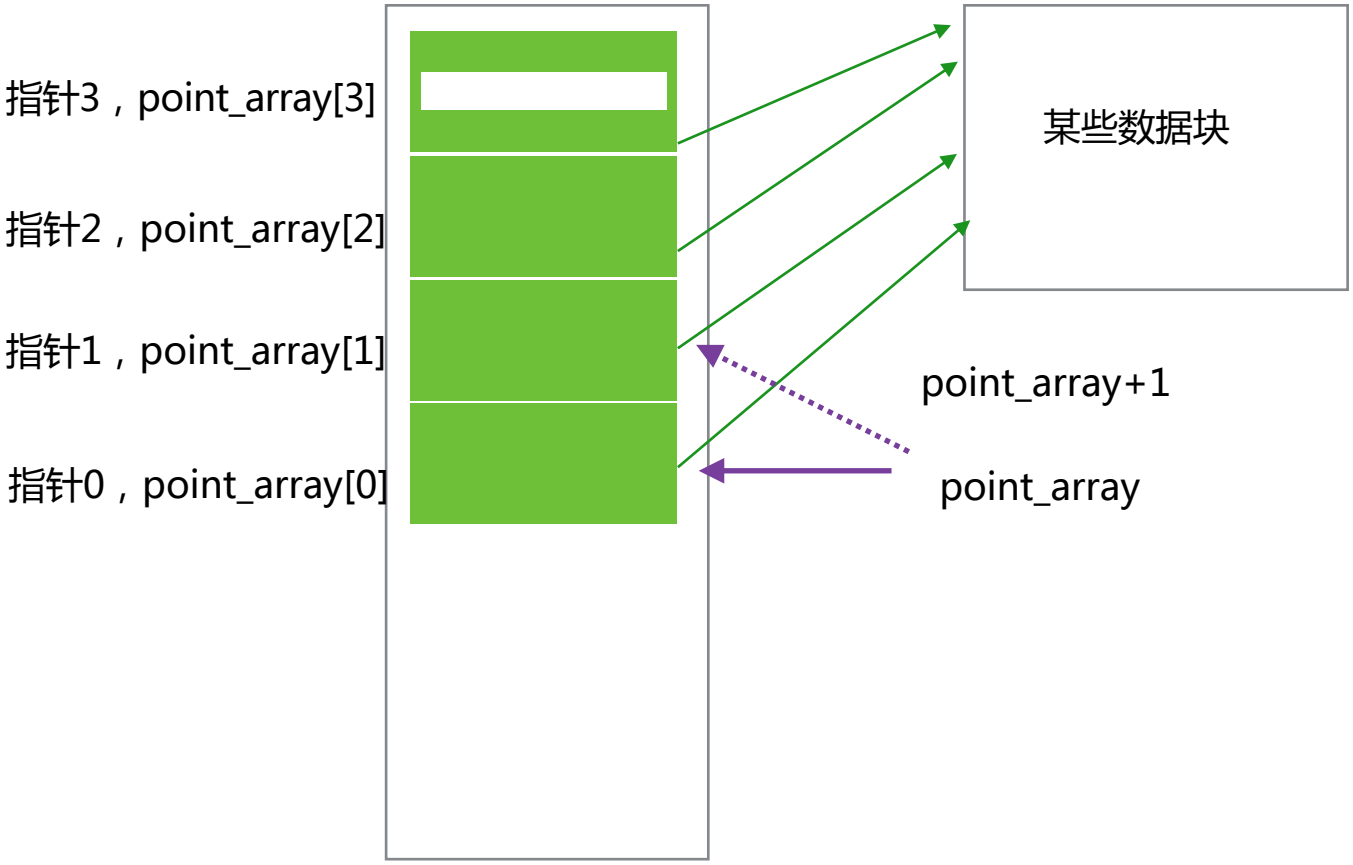
```c
#include <stdio.h>

/*
   typedef int(MYINT5)[5];
   typedef float(MYFLOAT10)[10];

   数组定义：

   MYINT5       iArray;  等价于 int iArray[5];
   MYFLOAT10    fArray;  等价于 float fArray[10];
 */


/*定义 数组类型，并用数组类型定义变量*/

int main(void)
{
    typedef int(MYINT5)[5];

    int i = 0;

    MYINT5 array;

    for (i=0; i<5; i++)
    {
        array[i] = i;
    }

    for (i=0; i<5; i++)
    {
        printf("%d ", array[i]);
    }

    return 0;
}
```

5.3 数组指针和指针数组

5. 3. 1 指针数组

```
char *point_array[4];
(char *)point_array[4];
```

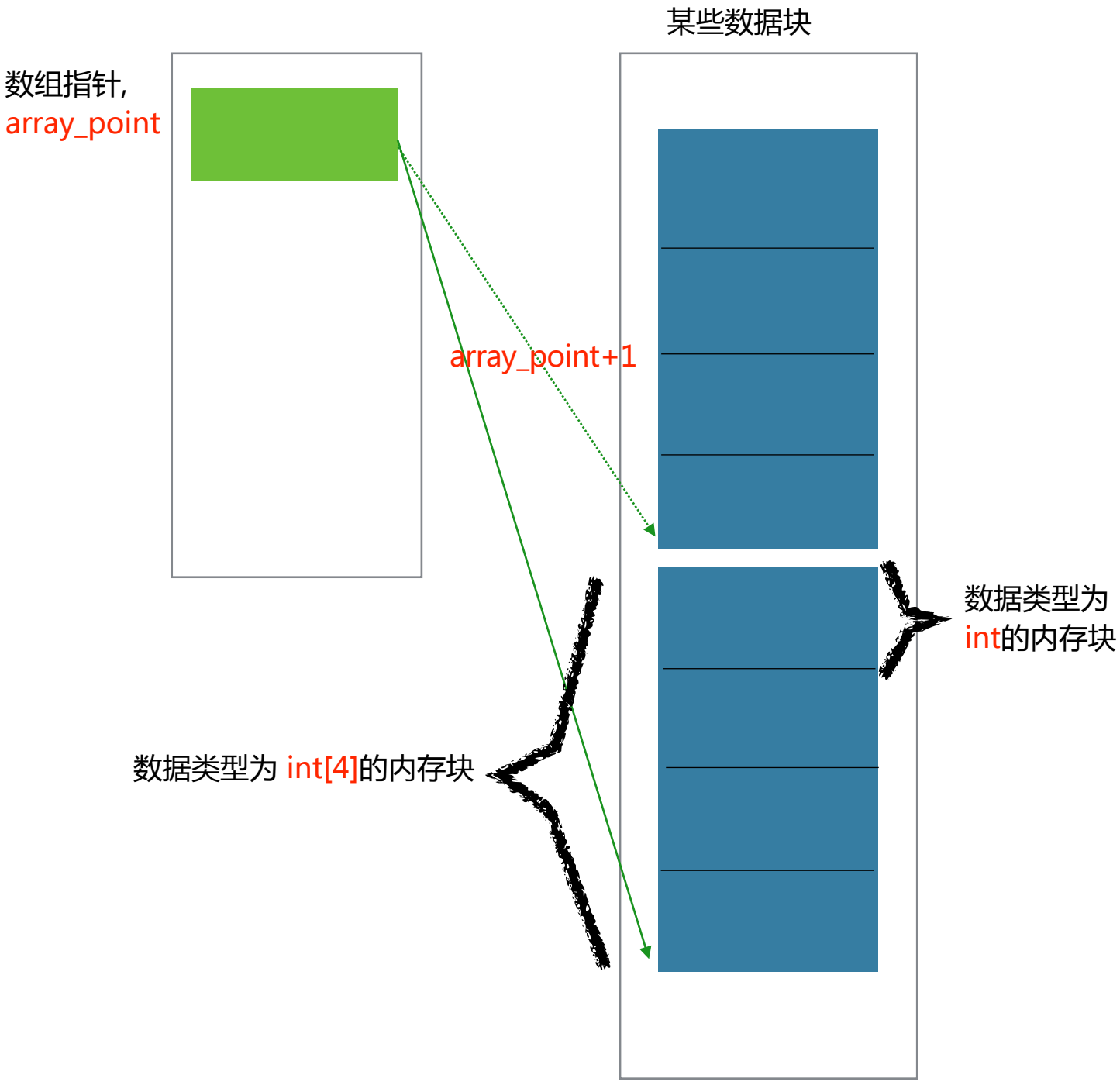
指针数组，是一个数组, 里面的每一个元素都是一个指针。（多个指针）



5.3.2 数组指针

```
int (*array_point)[4];
```

是一个指针，指向一个数组。（一个指针）



指针本是一种类型，但又说什么类型的指针，只不过是说指针所指向的数据是什么类型而已。那么指向数组类型的指针，就只好叫数组指针。

5.3.3 定义数组指针

```
int a[10];  //a的类型是一个指向int类型的指针。
```

数组名a是数组首元素的起始地址，但并不是数组的起始地址。

```
&a;         //&a的类型是一个指向数组int[10]类型的指针。
```

通过将取地址符&作用于数组名可以得到整个数组的起始地址。

1）通过数组类型定义数组指针:

```
typedef int(ArrayType)[5];  //定义类型ArrayType 为int[5]类型

ArrayType* pointer;         //那么指向ArrayType的指针就是指向int[5]类型的指针
```

2) 通过数组指针定义数组指针

```
typedef int (*MyPointer)[5];//定义类型MyPointer 为指向int[5]类型的指针

MyPointer myPoint;          //那么用这种类型的指针定义的便利都是指向int[5]类型的
```

3）直接定义

```
int (*pointer)[5];
```

5.4 多维数组名的本质

int a [3] [5] 的表示形式:

第0行第1列元素地址	a [0]+1	*a+1	&a[0] [1]
第1行第2列元素地址	a [1]+2	*(a+1)+2	&a[1] [2]
第 i 行第 j 列元素地址	a [i]+j	*(a+i)+j	&a[i] [j]
第1行第2列元素的值	*(a [1]+2)	*(*(a+1)+2)	a[1] [2]
第 i 行第 j 列元素的值	*(a [i]+j)	*(*(a+i)+j)	a[i] [j]

```c
#include <stdio.h>

int main(int)
{
    int a[3][5], i=0, j=0;

    // a 多维数组名 代表?
    printf("a %d , a+1:%d ", a, a+1);        //a+1的步长 是20个字节 5*4
    printf("&a %d , &a+1:%d ", &a, &a+1);    //&+1的步长 是5*4*3 = 120

    //多维数组名的本质就是一个指向低一个维度的数组的指针
    //步长为低一个维度的数据类型大小

    //定义一个指向数组的指针变量
    int (*pArray)[5] ;       //告诉编译器  分配 4个字节的内存 32bit平台下
    for (i=0; i<3; i++)
    {
        for (j=0; j<5; j++)
        {
            printf("%d ", pArray[i][j]);
        }
    }

    // (a+i)                代表是整个第i行的地址   二级指针
    // *(a+i)               代表 1级指针   第i行首元素的地址
    // *(a+i) + j  ===> &  (a[i][j])
    //*( *(a+i) + j) ===>   a[i][j]元素的值

    return 0;
}
```

我的意思是:一个指针是什么类型,它每次+1.就移动这个类型的总的字节数

比如:int *p; p+=1;则p移动4个字节,(都是64位编译环境来说.) 如果是char *c;p+=1移动的是一个字节.double *p,p+=1,移动的是8个字节

这个你赞同吗?

赞同。

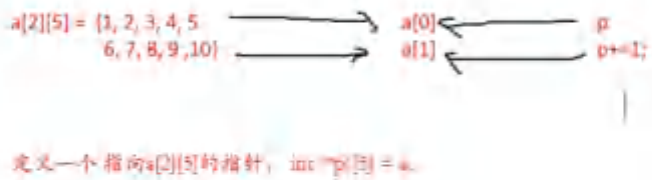
那么既然一个指针是什么类型,我就该移动这个类型的总字节长度. 那我int (*p)[3].如果表示的是二维数组指针类型,那我p+=1跨越的是这个二维数组的总字节长度对吗?

跨越的应该是一个int[3]
也就是4*12的长度

比如说我一个二维数组是int arr[2][3]={{1,2,3},{4,5,6}};那么它的总字节长度是24.

通过2*3*4得来

如果你认为int (*p)[3];表示的是一个二维数组指针的话,那么p+=1,应该移动24个字节



定义一个指向a[2][5]的指针, int (*p)[5] = a;

p 每次加的跨度 应该是 int(*p)[5] 中的5.

是5个int, 大小是5*20个字节

但实际是却是12个字节,所以我认为,int (*p)[3]是表示一个一维数组的指针,因为这个一维数组的字节数才是12个,如果要表示一个二维数组的指针应该是:int (*p)[2][3]; int (*p)[3]只是表示里面第一个元素的指针

我明白你的意思了

你是说 如果*p 操作 应该取什么类型的值 对码

*p 确实是一个一位数组， 就像 你 int*p = a[10], 那么你*p就是 一个数。

但是 int(*p)[3] 如果作为形参，是接受的实参应该为二维数组。比如， int a[2][3]; 然后 int foo(int (*p)[3]); 调用的时候应该是foo(a);

int(*p)[3] 应该是一个 一维数组的指针， 但是他应该作为二维数组的形参。 他指向的值 是一个一维数组。

所以操作二维数组， 应该用int(*p)[3]这种指针来操作。

数组作为参数传递的话传递的这个数组的地址还是第一个元素的地址?虽然这两个值一样,但意义不一样

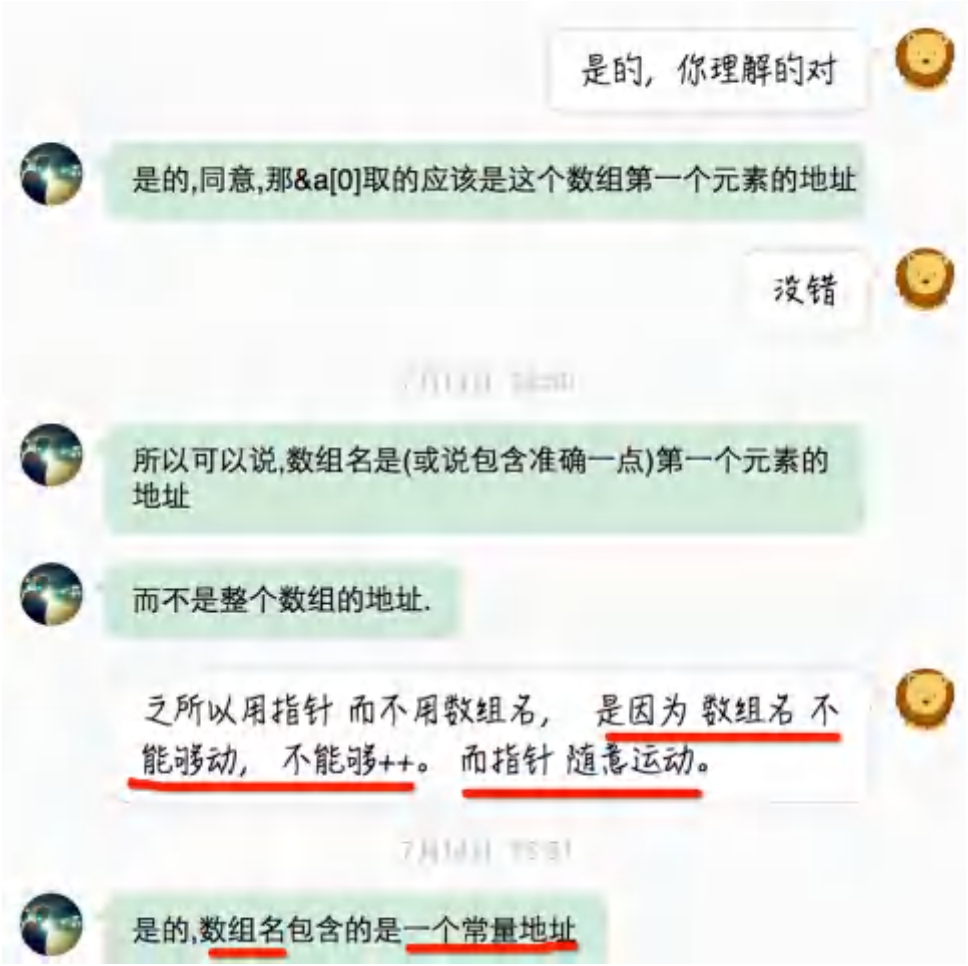
数组首元素的地址

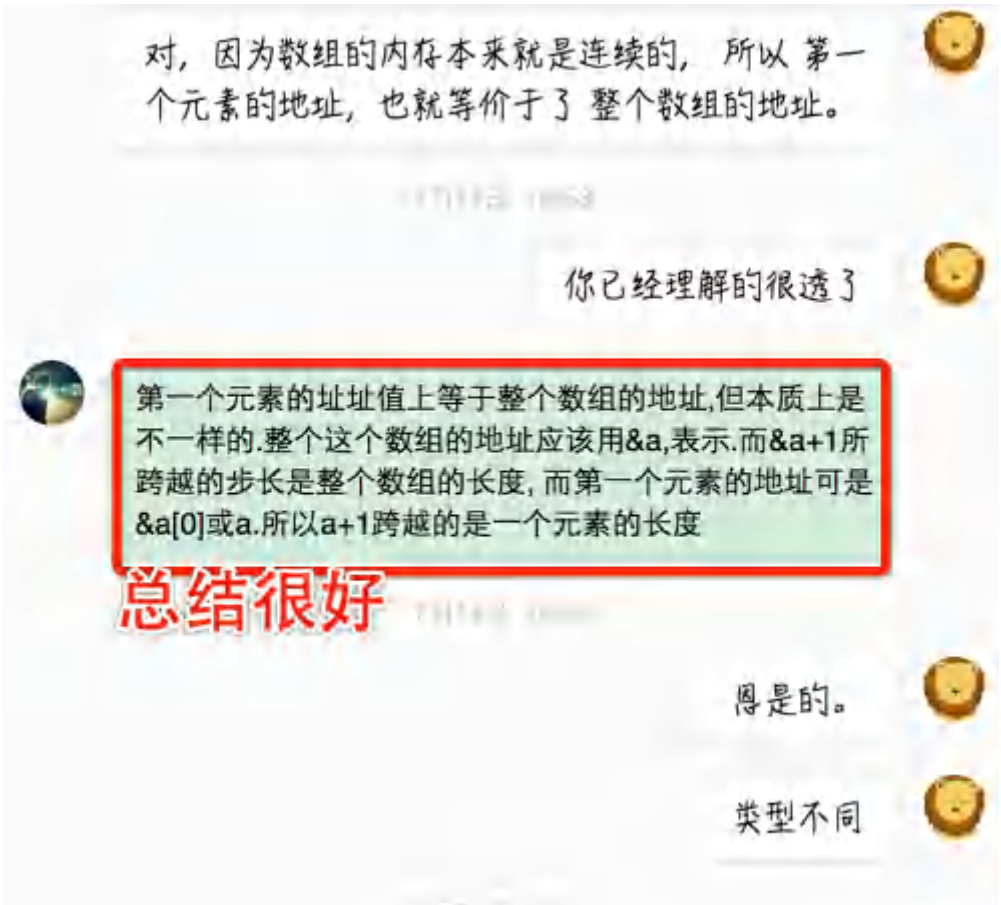
数组名字， 就是这个数组的首地址。

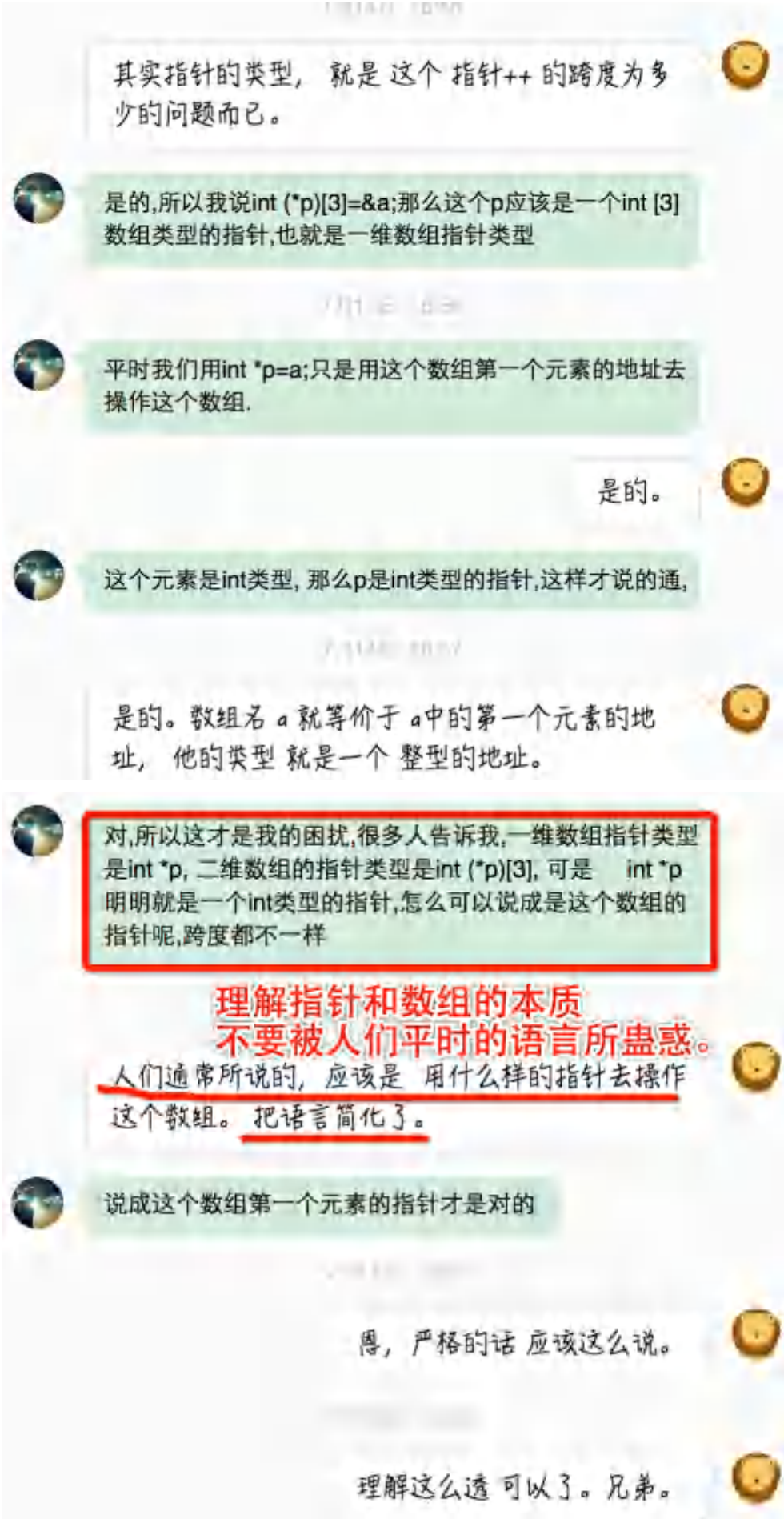
我赞同操作二维数组， 应该用int(*p)[3]这种指针来操作,但本质上应该是说.用的是这个二维数组的第一个元素的地址来操作,因为这样移动方变,不会越界,如果用整个二维数组的地址,那跨一步直接就跨出去了

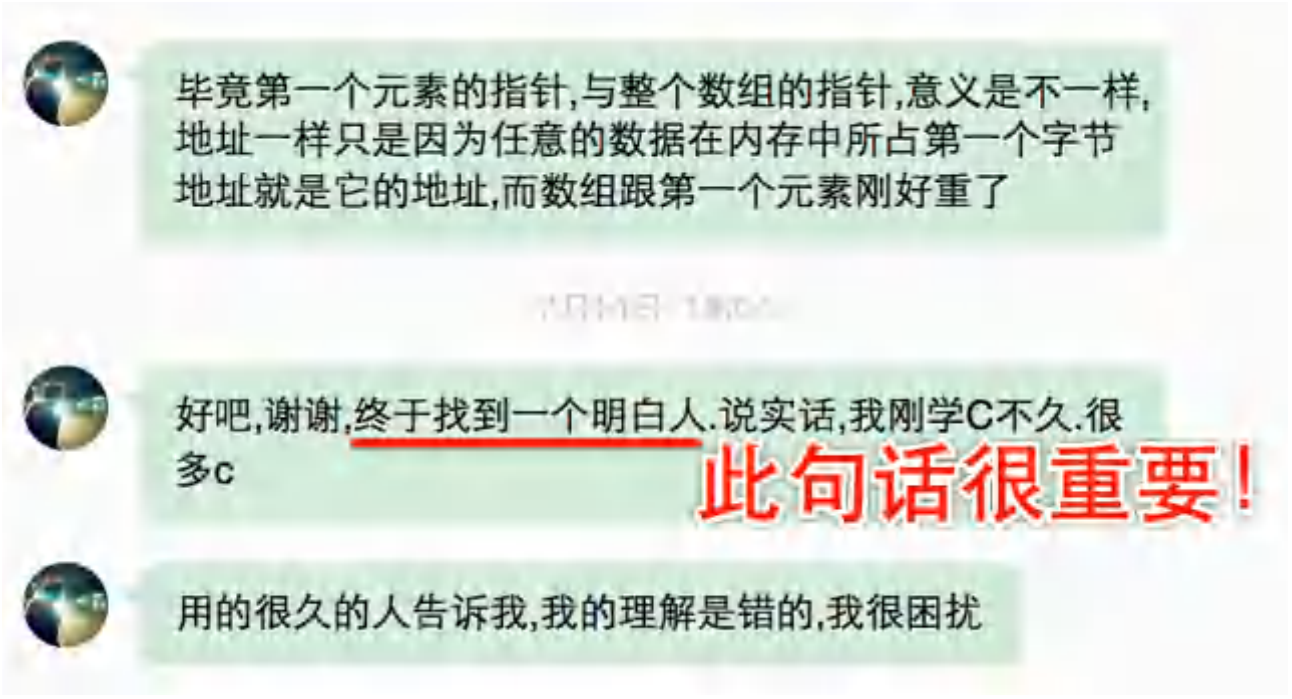
总结的很好

比如 int a[10]， 那么 a 等价于 &a[0]


是的，你理解的对
是的,同意,那&a[0]取的应该是这个数组第一个元素的地址
没错
所以可以说,数组名是(或说包含准确一点)第一个元素的地址
而不是整个数组的地址.
之所以用指针 而不用数组名， 是因为 数组名 不能够动， 不能够++。 而指针 随意运动。
是的,数组名包含的是一个常量地址


对，因为数组的内存本来就是连续的， 所以 第一个元素的地址，也就等价于了 整个数组的地址。
你已经理解的很透了
第一个元素的址址值上等于整个数组的地址,但本质上是不一样的.整个这个数组的地址应该用&a,表示.而&a+1所跨越的步长是整个数组的长度,而第一个元素的地址可是&a[0]或a.所以a+1跨越的是一个元素的长度
总结很好
嗯是的。
类型不同


其实指针的类型，就是这个指针++的跨度为多少的问题而已。
是的,所以我说int (*p)[3]=&a;那么这个p应该是一个int [3]数组类型的指针,也就是一维数组指针类型
平时我们用int *p=a;只是用这个数组第一个元素的地址去操作这个数组.
是的。
这个元素是int类型, 那么p是int类型的指针,这样才说的通,
是的。数组名a就等价于a中的第一个元素的地址，他的类型就是一个整型的地址。
对,所以这才是我的困扰,很多人告诉我,一维数组指针类型是int *p, 二维数组的指针类型是int (*p)[3], 可是 int *p明明就是一个int类型的指针,怎么可以说成是这个数组的指针呢,跨度都不一样
理解指针和数组的本质
不要被人们平时的语言所蛊惑。
人们通常所说的，应该是用什么样的指针去操作这个数组。把语言简化了。
说成这个数组第一个元素的指针才是对的
嗯，严格的话应该这么说。
理解这么透可以了。兄弟。


毕竟第一个元素的指针,与整个数组的指针,意义是不一样,
地址一样只是因为任意的数据在内存中所占第一个字节
地址就是它的地址,而数组跟第一个元素刚好重了
好吧,谢谢,终于找到一个明白人.说实话,我刚学C不久.很
多c
此句话很重要!
用的很久的人告诉我,我的理解是错的,我很困扰

5.5 多维数组的参数退化

5.5.1 多维数组的线性存储特性

```c
#include <stdio.h>

void printfArray(int *array, int size)
{
    int  i = 0;
    for (i=0; i<size; i++)
    {
        printf("%d ", array[i]);
    }
}
int main(void)
{
    int a[3][5];
    int i, j, tmp = 1;

    for (i=0; i<3; i++)
    {
        for (j=0; j<5; j++)
        {
            a[i][j] = tmp++;
        }
    }

    //把二维数组 当成  1维数组 来打印 证明线性存储
    printfArray((int *)a, 15);

    return 0;
}
```

5.5.2 多维数组的3种形式参数

```c
#include <stdio.h>

void printArray01(int a[3][5])
{
    int i, j;
    for (i=0; i<3; i++)
    {
        for (j=0; j<5; j++)
        {
            printf("%d ", a[i][j]);
        }
    }
}
```

```c
void printArray02(int a[][5])
{
    int i, j;
    for (i=0; i<3; i++)
    {
        for (j=0; j<5; j++)
        {
            printf("%d ", a[i][j]);
        }
    }
}

void printArray03( int (*b)[5])
{
    int i, j;
    for (i=0; i<3; i++)
    {
        for (j=0; j<5; j++)
        {
            printf("%d ", b[i][j]);
        }
    }
}

int main(void)
{
    int a[3][5], i=0, j=0;
    int tmp = 1;

    for (i=0; i<3; i++)
    {
        for (j=0; j<5; j++)
        {
            a[i][j] = tmp++;
        }
    }

    printArray03(a);

    return 0;
}
```

5.5.3 形参退化成指针的原因

原因1：高效
原因2：C语言处理a[n]的时候，它没有办法知道n是几，它只知道&n[0]是多少，它的值作为参数传递进去了，虽然c语言可以做到直接int fun(char a[20])，然后函数能得到20这个数字，但是，C没有这么做。

练习

找到数组中指定字符串的位置。

```
#define NUM(a) (sizeof(a)/sizeof(*a))

char* keywords[] = {
    "while",
    "case",
    "static",
    "do"
};

int searcheKeyTable(const char* table[], const int size,
                    const char* key, int *pos);
```

作业：

将字符串数组进行排序。

要求把第一种模型的结果 加上 第二种模型的结果 放在第三种模型中，并且排序。最后输出结果

```
#define IN
#define OUT

int sort(IN char **array1, IN int num1,
         IN char (*array2)[30], IN int num2,
         OUT char ***myp3, OUT int *num3);


int  main()
{
    int ret = 0;
    char *p1[] = {"aa", "ccccccc", "bbbbbb"};
    char buf2[10][30] = {"111111", "3333333", "222222"};
    char **p3 = NULL;
    int len1, len2, len3, i = 0;

    len1 = sizeof(p1)/sizeof(*p1);
    len2 = 3;

    ret = sort(p1, len1, buf2, len2, &p3, &len3);

    return 0;
}
```

6 宏

6.1 预处理步骤

代码在交给编译器编译的时候，首先需要进行预处理，代码展开动作。

1、把用\字符续行的多行代码接成一行。例如:

```
#define STR "hello, "\
                “world"
```

经过这个预处理步骤之后接成一行

```
#define STR "hello, " “world”
```

这种续行的写法要求\后面 紧跟换行,中间不能有其它空白字符。

2、把注释(不管是单行注释还是多行注释)都替换成一个空格。

3、经过以上两步之后去掉了一些换行,有的换行在续行过程中去掉了,有的换行在多行注释之中,也随着注释一起去掉了,剩下的代码行称为逻辑代码行。然后预处理器把逻辑代码行划分 成Token和空白字符,这时的Token称为预处理Token,包括标识符、整数常量、浮点数常量、 字符常量、字符串、运算符和其它符号。继续上面的例子,两个源代码行被接成一个逻辑代码行,然后这个逻辑代码行被划分成Token和空白字符:

```
#define STR "hello, " “world”
```

#, define, 空格, STR, 空格, "hello, ", Tab, Tab, "world"。

在划分Token时可能会遇到歧义,例如a+++++b这个表达式,既可以划分成a,++,++,+,b,也 可以划分成a,++,+,++,b。C语言规定按照从前到后的顺序划分Token,每个Token都要尽可 能长,所以这个表达式应该按第一种方式划分。其实按第一种方式划分Token是不合语法的,因 为++运算符的操作数必须是左值,如果a是左值则a++是合乎语法的,但a++这个表达式的值就不 再是左值了,所以a++再++就不合语法了,按第二种方式划分Token反倒是合乎语法的。即

便如 此,C编译器对这个表达式做词法分析时还是会按第一种方式划分Token,然后在语法和语义分析时再报错。

4、在Token中识别出预处理指示,做相应的预处理动作,如果遇到#include预处理指示,则把 相应的源文件包含进来,并对源文件做以上1-3步预处理。如果遇到宏定义则做宏展开。

一条预处理指示由一个逻辑代码行组成,以#开头,后面跟若干个预处理Token,在预处理指示中允许使用的空白字符只有空格和Tab。

5、找出字符常量或字符串中的转义序列,用相应的字节来替换它,比如把\n替换成字节0x0a。

6、把相邻的字符串连接起来。继续上面的例子,如果代码中有:

```
printf(
        STR);
```

经过第3步处理划分成以下Token:

printf, (, 换行, Tab, STR,), ;, 换行

经过第4步宏展 开后变成以下Token:

printf, (, 换行, Tab, "hello, “, Tab, Tab, “world",), ;, 换行

然后把相邻的字符串连接起来,变成以下Token:

printf, (, 换行, Tab, "hello, world”,), ;, 换行

7、经过以上处理之后,把空白字符丢掉,把Token交给C编译器做语法解析,这时就不再是预处理Token,而称为C Token了。这里丢掉的空白字符包括空格、换行、水平Tab、垂直Tab、 分页符。
继续上面的例子,最后交给C编译器做语法解析的Token是:

```
printf, (, "hello, world", ), ;
```

注意,把一个预处理指示写成多行要用\续行,因为根据定义,一条预处理指示 只能由一个逻辑代码行组成,而把C代码写成多行则不需要用\续行,因为换行在C代码中只不过是一种空白字符,在做语法解析时所有空白字符都已经丢掉了。

6.2 宏定义

6.2.1 函数式定义

```
#define MAX(a, b) ((a)>(b)?(a):(b))

k = MAX(a+1, b+1);
```

宏展开：

```
k = ((a+1)>(b+1)?(a+1):(b+1));
```

6.2.2 函数式宏定义 和 真正的函数调用有什么不同：

1、函数式宏定义的参数没有类型,预处理器只负责做形式上的替换,而不做参数类型检查,所以传参时要格外小心。

2、调用真正函数的代码和调用函数式宏定义的代码编译生成的指令不同。如果MAX是个真正的函数,那么它的函数体return a > b ? a : b;要编译生成指令,代码中出现的每次调用也要编译生成传参指令和call指令。而如果MAX 是个函数式宏定义,这个宏定义本身倒不必编译生成指令,但是代码中出现的每次调用编译生成的指令都相当于一个函数体,而不是简单的几条传参指令和call指令。所以,使用函数式宏定义编译生成的目标文件会比较大。

6.2.3 宏定义的陷阱

定义这种宏要格外小心,如果上面的定义写成

```
#define MAX(a, b) (a>b?a:b)
```

省去内层括号,则宏展开就成了

```
k = (a+1>b+1?a+1:b+1);
```

运算符的优先级就错了。

4、调用函数时先求实参表达式的值再传给形参,如果实参表达式有Side Effect,那么这些Side Effect只发生一次。
例如MAX(++a, ++b)
如果MAX是个真正的函数,a和b只增加一次。但如 果MAX是上面那样的宏定义,则要展开成k = ((++a)>(++b)?(++a):(++b)),a和b就不一定是增加一次还是两次了。

6.2.4 有关宏函数 do{…}while(0) 用法

看其他程序源码时,经常发现有宏定义用

```
#define XYZ \
    do{...} while(0)
```

这是一个奇怪的循环，它根本就只会运行一次，为什么不去掉外面的do{..}while结构呢？原来这也是非常巧妙的技巧。在工程中可能经常会引起麻烦，而上面的定义能够保证这些麻烦不会出现。下面是解释：

假设有这样一个宏定义

```
#define macro(condition) \
if(condition) dosomething()
```

现在在程序中这样使用这个宏：

```
if(temp)
    macro(i);
else
    doAnotherThing();
```

一切看起来很正常，但是仔细想想。这个宏会展开成：

```
if(temp)
    if(i) dosomething();
else
    doanotherthing();
```

整理后：

```
if(temp)
    if(i)
        dosomething();
    else
        doanotherthing();
```

这时的else不是与第一个if语句匹配，而是错误的与第二个if语句进行了匹配，编译通过了，但是运行的结果一定是错误的。

为了避免这个错误，我们使用do{....}while(0) 把它包裹起来，成为一个独立的语法单元，从而不会与上下文发生混淆。同时因为绝大多数的编译器都能够识别do{...}while(0) 这种无用的循环并进行优化，所以使用这种方法也不会导致程序的性能降低。

此外，这是为了含多条语句的宏的通用性。因为默认规则是宏定义最后是不能加分号的，分号是在引用的时候加上的。

为了看起来更清晰，这里用一个简单点的宏来演示：

```
#define SAFE_DELETE(p)  \
        do{             \
            free(p);    \
            p = NULL;   \
        } while(0)
```

假设这里去掉do...while(0),

```
#define SAFE_DELETE(p)    \
            free(p);      \
            p = NULL;
```

那么以下代码：

```
if(NULL != p)
    SAFE_DELETE(p)
else
    printf("p != NULL\n”);
```

宏定义展开如下：

```
if(NULL != p)
    free(p);
    p = NULL;
else
    printf("p != NULL\n”);
```

就有两个问题

1) 因为if分支后有两个语句，else分支没有对应的if，编译失败。

2) 假设没有else, SAFE_DELETE中的第二个语句无论if测试是否通过，会永远执行。

你可能发现，为了避免这两个问题，我不一定要用这个令人费解的do...while, 我直接用{}括起来就可以了

```
#define SAFE_DELETE(p)   {  \
            delete p;       \
            p = NULL;       \
        }
```

的确，这样的话上面的问题是不存在了，但是想对于C程序员来讲，在每个语句后面加分号是一种约定俗成的习惯，这样的话，以下代码:

```
if(NULL != p)
```

```
    SAFE_DELETE(p);
else
    printf("p != NULL\n”);
```

宏定义展开如下：

```
if(NULL != p)
{
    free(p);
    p = NULL;
};
else
    printf("p != NULL\n");
```

由于if后面的大括号后面加了一个分号，导致其else分支就无法通过编译了（原因同上），所以采用do...while(0)是做好的选择了。

```
#define SAFE_DELETE(p)  \
        do{             \
            free(p);    \
            p = NULL;   \
        } while(0)
```

那么

```
if(NULL != p)
    SAFE_DELETE(p);
else
    printf("p != NULL\n");
```

展开为

```
if(NULL != p)
do{
    free(p);
    p = NULL;
}while(0);
else
    printf("p != NULL\n");
```

一切正常。

6.2.5 #、##运算符和可变参数

✺宏#运算符

在函数式宏定义中,#运算符用于创建字符串,#运算符后面应该跟一个形参(中间可以有 空格 或 Tab),例如:

```
#include <stdio.h>

#define STR(s) #s

int main(void)
{
    char *str = NULL;

    str = STR(hello world);
    printf("%s\n", str);

    return 0;
}
```

输出结果“hello world” ，自动用"号把实参括起来成为一个字符串,并且实参中 的连续多个空白字符被替换成一个空格。

✺宏##运算符

在宏定义中可以用##运算符把前后两个预处理Token连接成一个预处理Token,和#运算符不 同,##运算符不仅限于函数式宏定义,变量式宏定义也可以用。例如:

```
#include <stdio.h>

#define CONCAT(a, b) a##b
//##拼接的不是字符串 而是Token

int main(void)
{
    char *concat = NULL;

    CONCAT(con, cat) = "abc";

    printf("%s\n", concat);

    return 0;
```

```
}
```

```
#define HASH_HASH # ## #
```

中间的##是运算符,宏展开时前后两个#号被这个运算符连接在一起。注意中间的两个空格是不 可少的,如果写成####,会被划分成##和##两个Token,而根据定义##运算符用于连接前后两个 预处理Token,不能出现在宏定义的开头或末尾,所以会报错。

我们知道printf函数带有可变参数,函数式宏定义也可以带可变参数,同样是在参数列表中 用... 表示可变参数。例如:

定义：

```
#define showlist(...) printf(#__VA_ARGS__)

#define report(test, ...) ((test)?printf(#test):\
        printf(__VA_ARGS__))
```

调用：

```
showlist(The first, second, and third items.);

report(x>y, "x is %d but y is %d", x, y);
```

预处理之后：

```
printf("The first, second, and third items.");

((x>y)?printf("x>y"): printf("x is %d but y is %d", x, y));
```

在宏定义中,可变参数的部分用__VA_ARGS__表示,实参中对应...的几个参数可以看成一个参数 替换到宏定义中__VA_ARGS__所在的地方。

```c
#include <stdio.h>

#define DEBUG(format, ...) \
    printf(format, ##__VA_ARGS__)

int main(void)
{
    int i = 10;

    DEBUG("i = %d\n", i);

    DEBUG("ABCDEF\n");

    return 0;
}
```

这个函数式宏定义可以这样调用:DEBUG("info no. %d", 1)。也可以这样调用:DEBUG("info")。后者相当于可变参数部分传了一个空参数,但展 开后并不是printf(“info” ,),而是printf(“info”),当__VA_ARGS__是空参数时,##运算符把它前 面的,号“吃”掉了。

作业：

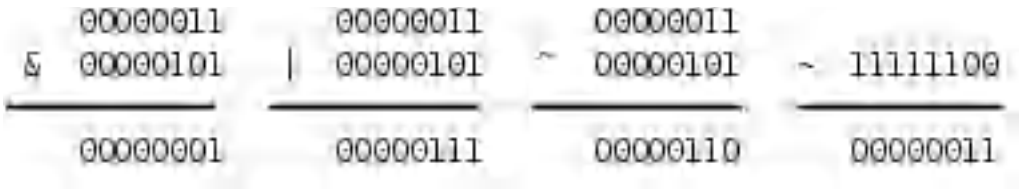
利用宏函数的可变参数封装一套自己个性的debug调试接口

7 位运算

整数在计算机中用二进制的位来表示,C语言提供一些运算符可以直接操作整数中的位,称为位 运算,这些运算符的操作数都必须是整型的。在以后的学习中你会发现,很多信息利用整数中 的某几个位来存储,要访问这些位,仅仅有对整数的操作是不够的,必须借助位运算。

7.1 按位与、或、异或、取反运算

C语言提供了按位与(Bitwise AND)运算符&、按位或(Bitwise OR)运算符|和按位取反(Bitwise NOT)运算符~,此外还有按位异或(Bitwise XOR)运算符^。



注意,&、|、^运算符都是要做Usual Arithmetic Conversion的,~运算符也要做Integer Promotion,所以在C语言中其实并不存在8位整数的位运算,操作数在做位运算之前都至少被提 升为int型了,上面用8位整数举例只是为了书写方便。比如:

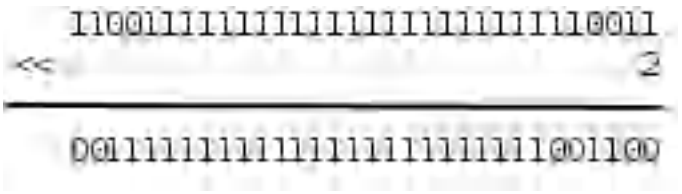
```
unsigned char c = 0xfc;
unsigned int i = ~c;
```

计算过程是这样的:常量0xfc是int型的,赋给c要转成unsigned char,值不变,c的十六进制表 示就是fc,计算~c时先提升为整型(000000fc)然后取反,最后结果是ffffff03。注意,如果 把~c看成是8位整数的取反,最后结果就得3了,这就错了。为了避免出错,一是尽量避免不同 类型之间的赋值,二是每一步计算都要按上一章讲的类型转换规则仔细检查。

7.2 位移运算

移位运算符(Bitwise Shift)包括左移<<和右移>>。左移将一个整数的各二进制位全部左移若干 位,

例如 0xcfffffff3<<2 得到0x3fffffcc:

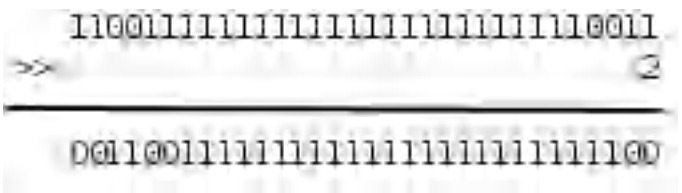


最高两位的11被移出去了,最低两位又补了两个0,其它位依次左移两位。但要注意,移动的位 数必须小于左操作数的总位数,比如上面的例子,左边是unsigned int 型,如果左移的位数大 于等于32位,则结果是未定义的。

在一定的取值范围内,将一 个整数左移1位相当于乘以2。比如二进制11(十进制3)左移一位变成110,就是6,再左移一 位变成1100,就是12。读者可以验证这条规律对负数也成立。当然,如果左移改变了符号位, 或者最高位是1被移出去了,那么结果肯定不是乘以2了,所以我说“在一定的取值范围内”。由于 计算机做移位比做乘法快得多,编译器可以利用这一点做优化,比如看到源代码中有i * 8,可 以编译成移位指令而不是乘法指令。

当操作数是无符号数时,右移运算的规则和左移类似,

例如0xcfffffff3>>2得到0x33fffffc:



最低两位的11被移出去了,最高两位又补了两个0,其它位依次右移两位。和左移类似,移动的 位数也必须小于左操作数的总位数,否则结果是Undefined的。在一定的取值范围内,将一个整 数右移1位相当于除以2,小数部分截掉。

当操作数是有符号数时,右移运算的规则比较复杂

如果是正数,那么高位移入0
如果是负数,那么高位移入1还是0不一定,这是Implementation-defined的。

综上所述,由于类型转换和移位等问题,使用有符号数做位运算是很不方便的,所以,建议只对无符号数做位运算,以减少出错的可能。

```c
#include <stdio.h>

int main(void)
{
    int i = 0xcffffff3;

    printf("%x\n", 0xcffffff3>>2);
    printf("%x\n", i>>2);

    return 0;
}
```

练习：

得到二进制中1的个数

7.3 掩码

如果要对一个整数中的某些位进行操作，怎样表示这些在整数中的位置呢？

可以用掩码（Mask）来表示，比如掩码0x0000ff00表示对一个32位整数的8-15位进行操作。

1. 取出8-15位

```
unsigned int a, b, mask = 0x0000ff00;

a = 0x12345678;
b = (a & mask) >> 8; /* 0x00000056 */
```

这样也可以达到同样的效果:

```
b = (a >> 8) & ~(~0 << 8);
```

2. 将8-15位清0.

```
unsigned int a, b, mask = 0x0000ff00;

a = 0x12345678;
b = a & ~mask; /* 0x12340078 */
```

3. 将8-15位置为1.

```
unsigned int a, b, mask = 0x0000ff00;

a = 0x12345678;
b = a | mask; /* 0x1234ff78 */
```

7.4 异或运算的一些特性

1、一个数和自己做异或的结果是0。如果需要一个常数0。

2、从异或的真值表可以看出,不管是0还是1,和0做异或值不变,和1做异或得到原值的相反 值。可以利用这个特性配合掩码实现某些位的翻转,

例如:

```
unsigned int a, b, mask = 1 << 6;

a = 0x12345678;
b = a ^ mask; /* 讲0-6位的值取反 */
```

3、x ^ x ^ y == y,因为x ^ x == 0,0 ^ y == y。这个性质有什么用呢?我们来看这样一个问题: 交换两个变量的值,不得借助于额外的存储空间,所以就不能采用temp = a; a = b; b = temp;的办法了。利用位运算可以这样做交换:

```
a = a ^ b;
b = b ^ a;
a = a ^ b;
```

分析一下这个过程。为了避免混淆,把a和b的初值分别记为a_0和b_0。第一行,a = a_0 ^ b_0;第 二行,把a的新值代入,得到b = b_0 ^ a_0 ^ b_0,等号右边的b_0相当于上面公式中的x,a_0相当 于y,所以结果为a_0;第三行,把a和b的新值代入,得到a = a_0 ^ b_0 ^ a_0,结果为b_0。

练习:

从无符号型x的第p位开始, 取n位。 最低位是第0位

```
unsigned int getBit(unsigned int x, int pos, int n);
```

8. 指针刺向结构体

8.1 结构体的类型和定义

结构体是一种构造数据类型.

用途：把不同类型的数据组合成一个整体-------自定义数据类型

```
#include <stdio.h>
#include <string.h>

//声明一个结构体类型
struct _Teacher
{
    char    name[32];
    char    tile[32];
    int     age;
    char    addr[128];
};

//定义结构体变量的方法
/*
   1)定义类型 用类型定义变量
   2)定义类型的同时，定义变量;
   3）直接定义结构体变量;
 */

struct _Student
{
    char    name[32];
    char    tile[32];
    int     age;
    char    addr[128];
}s1, s2;  //定义类型的同时，定义变量;

struct
{
    char    name[32];
    char    tile[32];
    int     age;
    char    addr[128];
}s3,s4; //直接定义结构体变量

//初始化结构体变量的几种方法
//1)
struct _Teacher t4 = {"name2", "tile2", 2, "addr2"};
//2)
struct Dog1
{
    char    name[32];
```

```
    char    tile[32];
    int     age;
    char    addr[128];
}d5 = {"dog", "gongzhu", 1, "ddd"};

//3)
struct
{
    char    name[32];
    char    tile[32];
    int     age;
    char    addr[128];
}d6 = {"dog", "gongzhu", 1, "ddd"};
//结构体变量的引用

int main(void)
{
    //struct _Teacher t1, t2;
    //定义同时初始化
    struct _Teacher t3 = {"name2", "tile2", 2, "addr2"};
    printf("%s\n", t3.name);
    printf("%s\n", t3.tile);

    //用指针法和变量法分别操作结构体
    struct _Teacher t4;
    struct _Teacher *pTeacher = NULL;
    pTeacher = &t4;

    strcpy(t4.name, "zhangsan");

    strcpy(pTeacher->addr, "ddddd");

    printf("t4.name:%s\n", t4.name);

    return 0;
}
```

8.2 结构体的赋值

```
#include <stdio.h>
#include <string.h>

//声明一个结构体类型
struct _Mytercher
{
    char    name[32];
    char    title[32];
    int     age;
    char    addr[128];
};
```

```c
typedef struct _Mytercher teacher_t;

void show_teacher(teacher_t t)
{
    printf("name : %s\n", t.name);
    printf("title : %s\n", t.title);
    printf("age : %d\n", t.age);
    printf("addr : %s\n", t.addr);
}

void  copyTeacher(teacher_t to, teacher_t from )
{
    to = from;
}

void  copyTeacher2(teacher_t *to, teacher_t *from )
{
    *to = *from;
}

//结构体赋值和实参形参赋值行为研究
int main(void)
{
    teacher_t t1, t2, t3;
    memset(&t1, 0, sizeof(t1));

    strcpy(t1.name, "name");
    strcpy(t1.addr, "addr");
    strcpy(t1.title, "title");
    t1.age = 1;

    show_teacher(t1);

    //结构体直接赋值
    t2 = t1;
    show_teacher(t2);

    copyTeacher(t3, t2);
    show_teacher(t3);

    copyTeacher2(&t3, &t2);
    show_teacher(t3);

    return 0;
}
```

8.3 结构体数组

```c
#include <stdio.h>
#include <string.h>

//声明一个结构体类型
struct _Mytercher
{
    char    name[32];
    char    title[32];
    int     age;
    char    addr[128];
};

typedef struct _Mytercher teacher_t;

void show_teacher(teacher_t t)
{
    printf("name : %s\n", t.name);
    printf("title : %s\n", t.title);
    printf("age : %d\n", t.age);
    printf("addr : %s\n", t.addr);
}

//定义结构体数组
int main(void)
{
    int i = 0;
    struct _Mytercher teaArray[3];

    for (i=0; i<3; i++)
    {
        strcpy(teaArray[i].name, "a");
        teaArray[i].age = i+20;
        strcpy(teaArray[i].title, "title");
        strcpy(teaArray[i].addr, "addr");
        show_teacher(teaArray[i]);
    }

    return 0;
}
```

8.4 结构体作为函数参数

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef struct Teacher
{
    char name[64];
    char *alisname;
    int age ;
    int id;
}Teacher;

void printTeacher(Teacher *array, int num)
{
    int i = 0;

    for (i=0; i<num; i++)
    {
        printf("\n================\n");
        printf("name : %s\n", array[i].name);
        printf("alisname : %s\n", array[i].alisname);
        printf("age:%d \n", array[i].age);
    }
}

void sortTeacer(Teacher *array, int num)
{
    int     i,j;
    Teacher tmp;

    for (i=0; i<num; i++)
    {
        for (j=i+1; j<num; j++)
        {
            if (array[i].age > array[j].age)
            {
                tmp = array[i];  //=号操作  赋值操作
                array[i] = array[j];
                array[j] = tmp;
            }
        }
    }
}

Teacher * createTeacher01(int num)
{
    Teacher * tmp = NULL;
    tmp = (Teacher *)malloc(sizeof(Teacher) * num); //  Teacher     Array[3]
    if (tmp == NULL)
    {
        return NULL;
    }
```

```
    return tmp; //

}

int createTeacher02(Teacher **pT, int num)
{
    int i = 0;
    Teacher * tmp = NULL;
    tmp = (Teacher *)malloc(sizeof(Teacher) * num); //  Teacher     Array[3]
    if (tmp == NULL)
    {
        return -1;
    }
    memset(tmp, 0, sizeof(Teacher) * num);

    for (i=0; i<num; i++)
    {
        tmp[i].alisname = (char *)malloc(60);
    }

    *pT = tmp;  //二级指针 形参 去间接的修改 实参 的值
    return 0; //
}

void FreeTeacher(Teacher *p, int num)
{
    int i = 0;
    if (p == NULL)
    {
        return;
    }
    for (i=0; i<num; i++)
    {
        if (p[i].alisname != NULL)
        {
            free(p[i].alisname);
        }
    }
    free(p);
}

int main(void)
{
    int         ret = 0;
    int         i = 0;
    int         num = 3;

    Teacher *pArray = NULL;

    ret = createTeacher02(&pArray, num);
    if (ret != 0)
    {
        printf("func createTeacher02() er:%d \n ", ret);
        return ret;
    }
```

```
    for (i=0; i<num; i++)
    {
        printf("\nplease enter age:");
        scanf("%d", & (pArray[i].age) );

        printf("\nplease enter name:");
        scanf("%s",  pArray[i].name ); //向指针所指的内存空间copy数据

        printf("\nplease enter alias:");
        scanf("%s",  pArray[i].alisname );  //向指针所指的内存空间copy数据
    }


    printTeacher(pArray, num);

    sortTeacer(pArray, num);

    printf("排序之后\n");

    printTeacher(pArray, num);

    FreeTeacher(pArray, num);

    return 0;
}
```

8.5 结构体嵌套指针

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef struct Teacher
{
    int id;
    char *name;
    char **student_name;
    int stu_num;
}Teacher;

void printTeacher(Teacher *array, int num)
{
    int i = 0;
    int j = 0;

    for (i=0; i<num; i++)
    {
        printf("\n--------------------\n");
        printf("teacher'name:%s\n", array[i].name);
        printf("id:%d \n", array[i].id);
```

```
        printf("student's count:%d\n", array[i].stu_num);

        for (j = 0; j < array[i].stu_num; j++) {
            printf("\tstudent[%d]:%s\n", j, array[i].student_name[j]);
        }
    }
}

void sortTeacer(Teacher *array, int num)
{
    int     i,j;
    Teacher tmp;

    for (i=0; i<num; i++)
    {
        for (j=i+1; j<num; j++)
        {
            if (array[i].id > array[j].id)
            {
                tmp = array[i];  //=号操作  赋值操作
                array[i] = array[j];
                array[j] = tmp;
            }
        }
    }
}

int createStudents(Teacher *t, int stunum)
{
    char **p = NULL;
    int i = 0;

    //二级指针的第三种内存模型
    p = (char **)malloc(stunum * sizeof(char *));  //打造二维内存
    for (i=0; i<stunum; i++)
    {
        p[i] = (char *)malloc(120);
    }

    t->student_name = p;

    return 0;
}

int createTeacher02(Teacher **pT, int num)
{
    int i = 0;
    Teacher * tmp = NULL;

    tmp = (Teacher *)malloc(sizeof(Teacher) * num);
    if (tmp == NULL)
    {
        return -1;
    }
    memset(tmp, 0, sizeof(Teacher) * num);
```

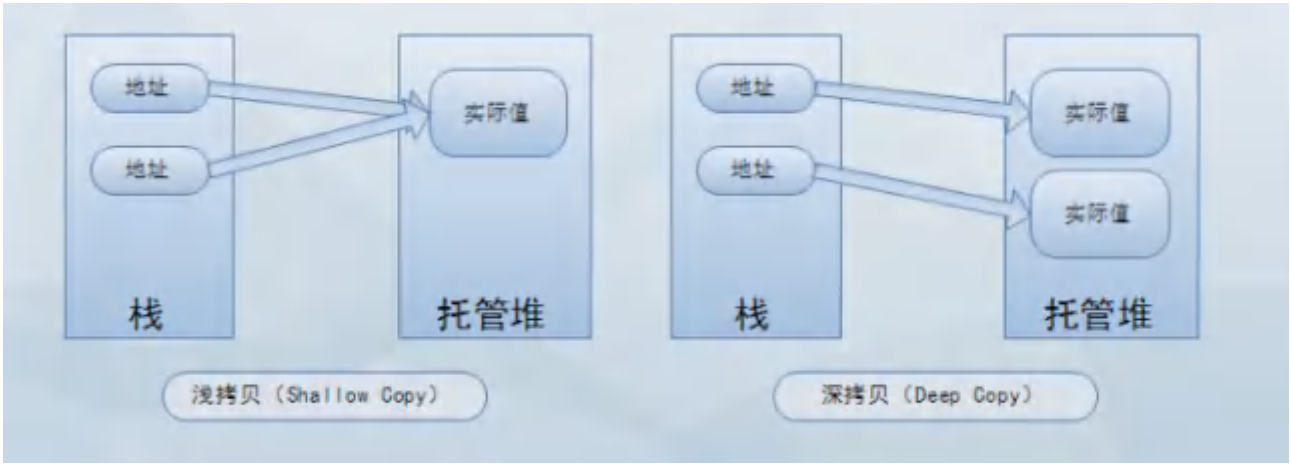
```
    for (i=0; i<num; i++)
    {

        //malloc一级指针
        tmp[i].name = (char *)malloc(60);

    }

    *pT = tmp;  //二级指针 形参 去间接的修改 实参 的值

    return 0;
}

void FreeTeacher(Teacher *p, int num)
{
    int i = 0, j = 0;
    if (p == NULL)
    {
        return;
    }
    for (i=0; i<num; i++)
    {
        //释放一级指针
        if (p[i].name != NULL)
        {
            free(p[i].name);
        }

        //释放二级指针
        if (p[i].student_name != NULL)
        {
            char **myp = p[i].student_name ;
            for (j=0; j<p[i].stu_num; j++)
            {
                if (myp[j] != NULL)
                {
                    free(myp[j]);
                }
            }
            free(myp);
            p[i].student_name = NULL;
        }
    }
    free(p);
}

int main(void)
{
    int         ret = 0;
    int         i = 0, j = 0;
    int         num = 3;
    Teacher *pArray = NULL;

    ret = createTeacher02(&pArray, num);
```

```
    if (ret != 0)
    {
        printf("func createTeacher02() er:%d \n ", ret);
        return -1;
    }

    for (i=0; i<num; i++)
    {
        printf("\nplease enter id:");
        scanf("%d", & (pArray[i].id) );

        printf("\nplease enter name:");
        scanf("%s",  pArray[i].name );

        printf("\nplease enter student number:");
        scanf("%d",  &(pArray[i].stu_num));

        //之所以在这开辟学生名称的内存 是因为在这里才知道这个老师
        //对应的学生数目
        createStudents(&pArray[i], pArray[i].stu_num);

        for (j=0; j<pArray[i].stu_num; j++)
        {
            printf("please enter student name:");
            scanf("%s",pArray[i].student_name[j] );
        }
    }
    printTeacher(pArray, num);

    sortTeacer(pArray, num);

    printf("排序之后\n");
    printTeacher(pArray, num);

    FreeTeacher(pArray, num);

    return 0;
}
```

练习

重写以上代码讲ID排序改为按照老师姓名排序

8.6 有关浅拷贝深拷贝问题




```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef struct Teacher
{
    char name[64];
    int age ;
    char *pname2;

}Teacher_t;
//结构体中套一个 1级指针 或 二级指针


//编译器的=号操作,只会把指针变量的值,从from  copy 到 to,但
//不会 把指针变量 所指 的 内存空间 给copy过去..//浅copy


//如果 想执行深copy,再显示的分配内存
void deepCopyTeacher(Teacher_t *to, Teacher_t *from)
{
    *to = *from;

    to->pname2 = (char *)malloc(100);
    strcpy(to->pname2, from->pname2);

    //memcpy(to, from , sizeof(Teacher_t));
}

//浅拷贝
void copyTeacher(Teacher_t *to, Teacher_t *from)
{
    memcpy(to, from , sizeof(Teacher_t));
```

```
    // or
    //*to = *from;
}

int main(void)
{
    Teacher_t t1;
    Teacher_t t2;

    strcpy(t1.name, "name1");
    t1.pname2 = (char *)malloc(100);
    strcpy(t1.pname2, "ssss");

    //t1 copy t2

    deepCopyTeacher(&t2, &t1);

    if (t1.pname2 != NULL)
    {
        free(t1.pname2);
        t1.pname2 = NULL;
    }

    if (t2.pname2 != NULL)
    {
        free(t2.pname2);
        t2.pname2 = NULL;
    }

    return 0;
}
```

8.7 结构体成员偏移量

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//一旦结构体定义下来,则结构体中的成员..内存布局 就定下了
typedef struct Teacher
{
    char name[64];  //64
    int age ;       //4
    int p;          //4

} Teacher_t;


int main(void)
{
    Teacher_t  t1;
    Teacher_t *p = NULL;
    p = &t1;

    int offsize1 =   (int)&(p->age) - (int)p;       //age 相对于结构体 Teacher的偏移量
    int offsize2 = (int )&(((Teacher_t *)0)->age );//绝对0地址 age的偏移量
    printf("offsize1:%d \n", offsize1);
    printf("offsize2:%d \n", offsize2);

    return 0 ;
}
```

8.8 有关结构体字节对齐

在用sizeof运算符求算某结构体所占空间时，并不是简单地将结构体中所有元素各自占的空间相加，这里涉及到内存字节对齐的问题。从理论上讲，对于任何 变量的访问都可以从任何地址开始访问，但是事实上不是如此，实际上访问特定类型的变量只能在特定的地址访问，这就需要各个变量在空间上按一定的规则排列， 而不是简单地顺序排列，这就是内存对齐。

内存对齐的原因：

1)某些平台只能在特定的地址处访问特定类型的数据；

2)提高存取数据的速度。比如有的平台每次都是从偶地址处读取数据，对于一个int型的变量，若从偶地址单元处存放，则只需一个读取周期即可读取该变量；但是若从奇地址单元处存放，则需要2个读取周期读取该变量。

结构体成员的内存分配规律是这样的：

从结构体的首地址开始向后依次为每个成员寻找第一个满足条件的首地址x，该条件是x % N = 0，并且整个结构的长度必须为各个成员所使用的对齐参数中最大的那个值的最小整数倍,不够就补空字节。

结构体中所有成员的对齐参数N的最大值称为结构体的对齐参数。

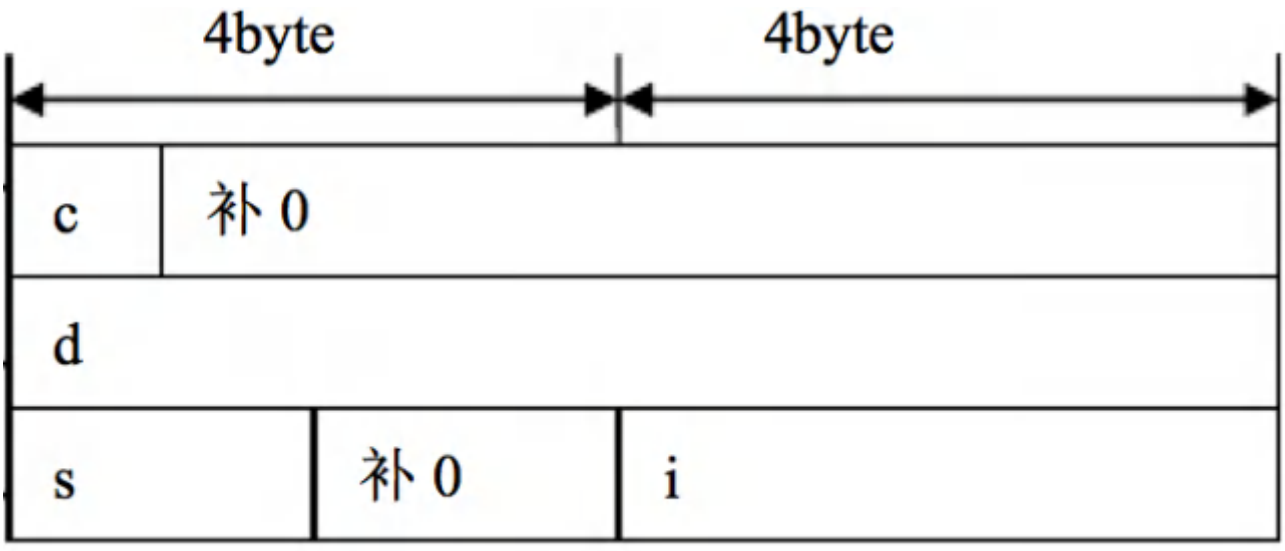
字节对齐可以程序控制，采用指令：

```
#pragma pack(xx)

#pragma pack(1)      //1字节对齐
#pragma pack(2)      //2字节对齐
#pragma pack(4)      //4字节对齐
#pragma pack(8)      //8字节对齐
#pragma pack(16)     //16字节对齐
```

```
#include <stdio.h>
```

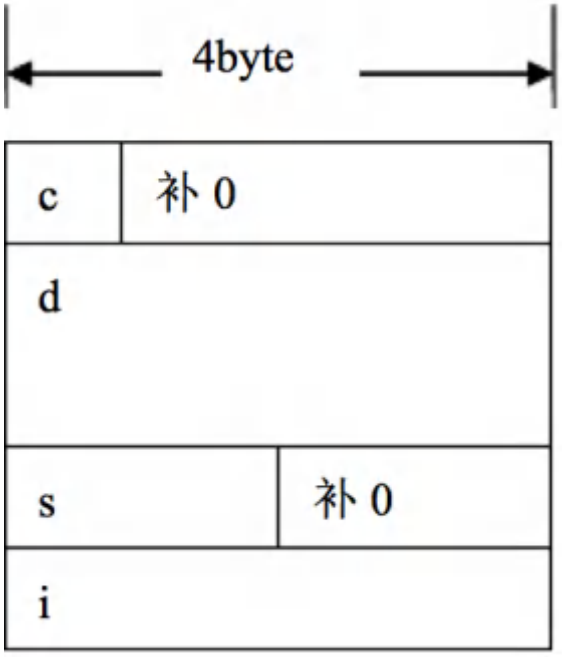
```
struct A
{
    char        c;      //1byte
    double      d;      //8byte
    short       s;      //2byte
    int         i;      //4byte
};

int main(int argc, char*argv[])
{
    struct A strua;

    printf("len: %d\n",sizeof(struct A));
    printf("char &c:%d\n"
           "double &d:%d\n"
           "short &s:%d\n"
           "int &i%d\n",
           &strua.c,
           &strua.d,
           &strua.s,
           &strua.i);

    return 0;
}
```

当系统以8个字节对齐时候



len的结果为24

当系统以4个字节对齐的时候

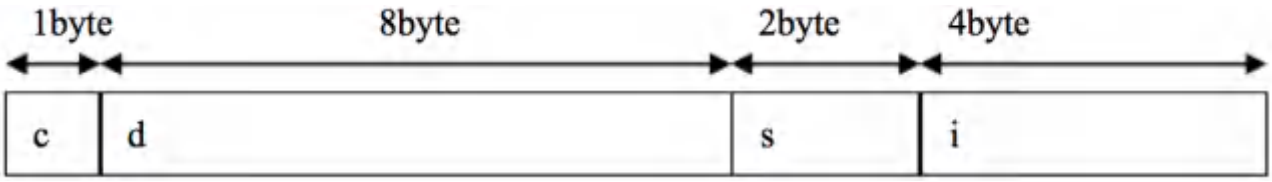


len的结果为20

当系统以2个字节字节对齐的时候

len的结果为16

当系统以1个字节字节对齐的时候

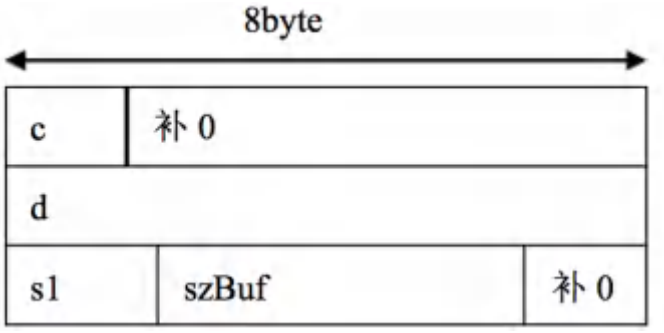


len的结果为15

有关成员数组的字节对齐

```
#include <stdio.h>

struct A

{
    char    c;    //1byte
    double  d;    //8byte
    short   s;     //2byte
    char    szBuf[5];
};

int main(int argc, char*argv[])
{

    printf("len: %d\n",sizeof(struct A));

    return 0;
}
```

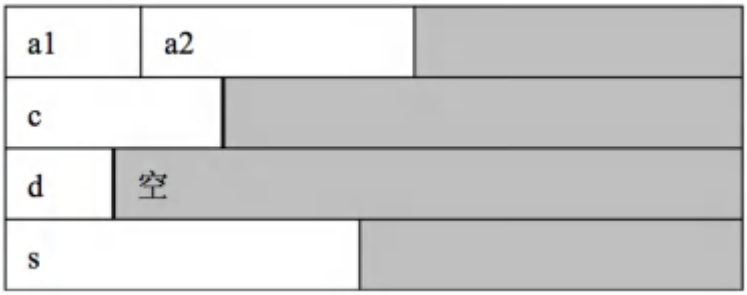
当结构体成员为数组时，并不是将整个数组当成一个成员来对待，而是将数组的每个元素当一个成员来分配，其他分配规则不变.
len的结果为24



有关不完整类型的字节对齐

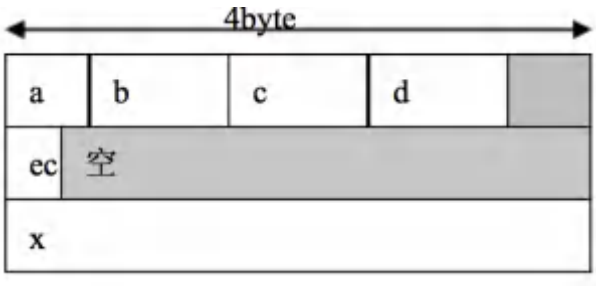
```
include <stdio.h>

struct A
{
    int     a1:5;
    int     a2:9;
    char    c;
    int     b:4;
    short   s;
};

int main(int argc, char*argv[])
{

    printf("len: %d\n",sizeof(struct A));

    return 0;
}
```

对于位段成员，存储是按其类型分配空间的，如int 型就分配4个连续的存储单元，如果是相邻的同类型的段位成员就连续存放，共用存储单元，此处如a1,a2将公用一个4字节的存储单元，当该类型的长度不够用时，就另起一个该类型长度的存储空间。有位段时的对齐规则是这样：同类型的、相邻的可连续在一个类型的存储空间中存放的位段成员作为一个该类型的成员变量来对待，不是同类型的、相邻的位段成员，分别当作一个单独得该类型的成员来对待，分配一个完整的类型空间，其长度为该类型的长度，其他成员的分配规则不变，仍然按照前述的对齐规则进行。

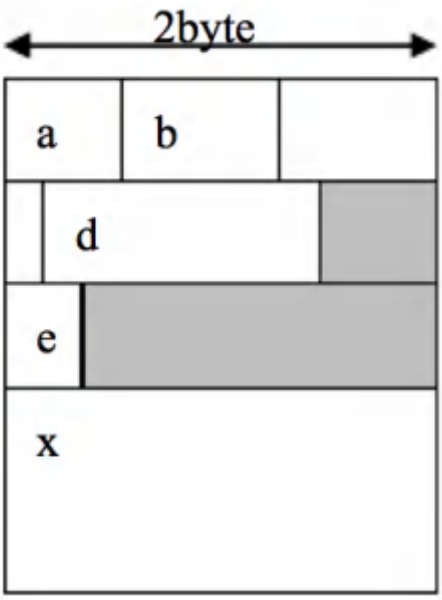



```
include <stdio.h>

struct  A
{
    int   a:5;
    int   b:7;
    int   c:6;
    int   d:9;
    char  e:2;
    int   x;
};

int main(int argc, char*argv[])
{

    printf("len: %d\n",sizeof(struct A));

    return 0;
}
```

Vc++6.0的对齐参数设置为8、16、4字节对齐时，sizeof(A)=12内存分布为：（灰色部分未使用）



当对齐参数设置为2字节时：（灰色部分未使用）sizeof(A)=10


2byte
a
b
d
e
x

练习题

阿里笔试题：

```
struct s1
{
        int i:8;
        int j:4;
        int a:3;
        double b;
}
struct s2
{
        int i:8;
        int j:4;
        double b;
        int a:3;
}
```

sizeof(s1)和sizeof(s2)为（ ）

A.20,20 B.16,24 C.16,20 D.12,16

IBM笔试题

```
struct{
        short a1;
        short a2;
        short a3;
}A;

struct{
        long a1;
        short a2;
}B;
```

注：sizeof(short)=2,sizeof(long)=4

sizeof(A)= ? , sizeof(B)= ? ,为什么？

9. 文件操作

为什么要使用文件?

输入数据

C语言执行程序

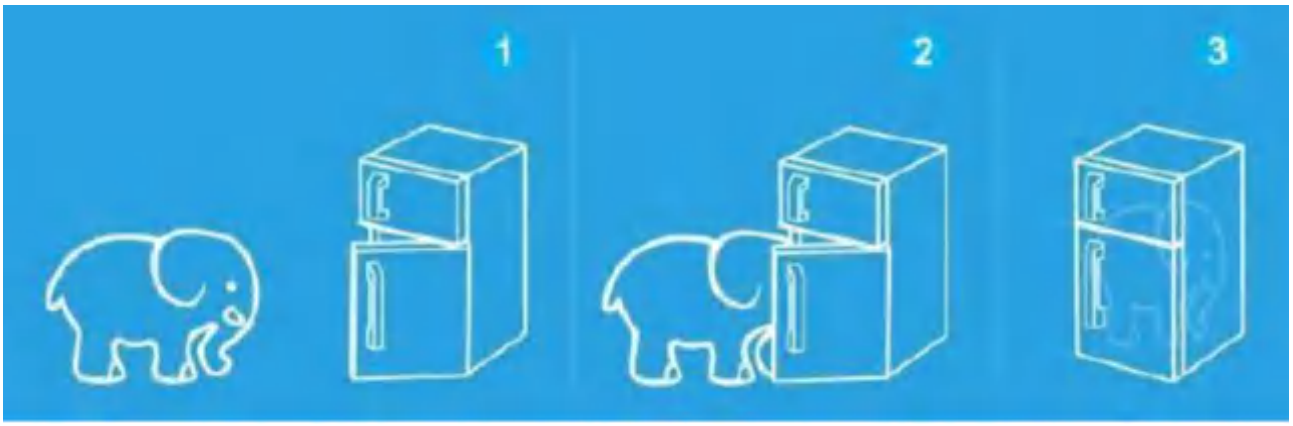
存在问题：运行完毕，结果消失！

输出数据

运行结果能否长期保存呢？

9.1 文件操作步骤

要把大象装进冰箱总共分几步?



对文件的操作步骤

1)引入头文件(stdio.h)

2)定义文件指针

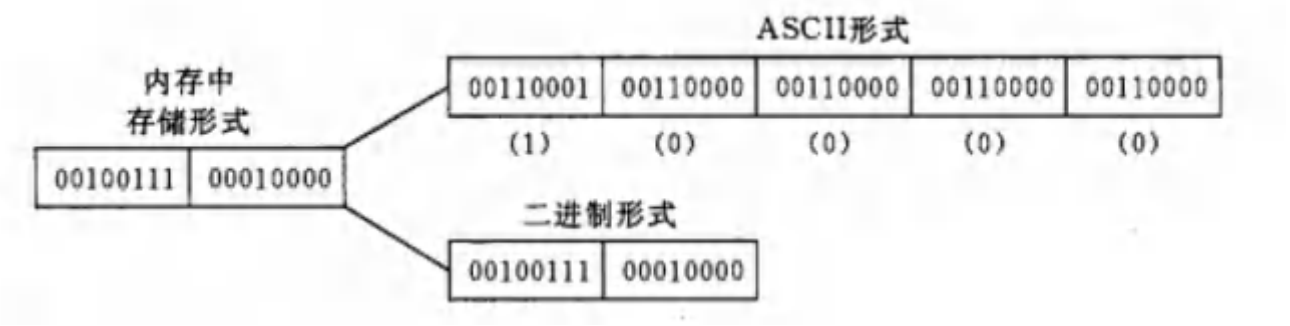
3)打开文件

4)文件读写

5)关闭文件

9.2 有关文件的概念

- 按文件的逻辑结构：
  - 记录文件：由具有一定结构的记录组成（定长和不定长）
  - 流式文件：由一个个字符（字节）数据顺序组成
- 按存储介质：
  - 普通文件：存储介质文件（磁盘、磁带等）
  - 设备文件：非存储介质（键盘、显示器、打印机等）
- 按数据的组织形式：
  - 文本文件： ASCII文件，每个字节存放一个字符的ASCII码
  - 二进制文件：数据按其在内存中的存储形式原样存放



✺ 流概念

流是一个动态的概念，可以将一个字节形象地比喻成一滴水，字节在设备、文件和程序之间的传输就是流，类似于水在管道中的传输，可以看出，流是对输入输出源的一种抽象，也是对传输信息的一种抽象。通过对输入输出源的抽象，屏蔽了设备之间的差异，使程序员能以一种通用的方式进行存储操作，通过对传输信息的抽象，使得所有信息都转化为字节流的形式传输，信息解读的过程与传输过程分离。

C语言中，I/O操作可以简单地看作是从程序移进或移出字节，这种搬运的过程便称为流(stream)。程序只需要关心是否正确地输出了字节数据，以及是否正确地输入了要读取字节数据，特定I/O设备的细节对程序员是隐藏的。

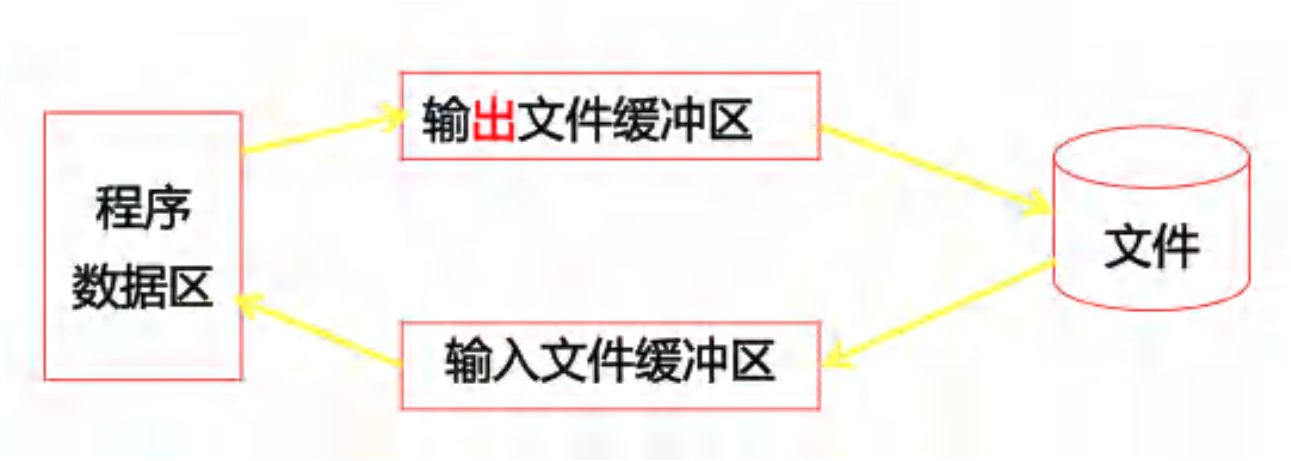
✺ 文件处理方法

1)文件缓冲区

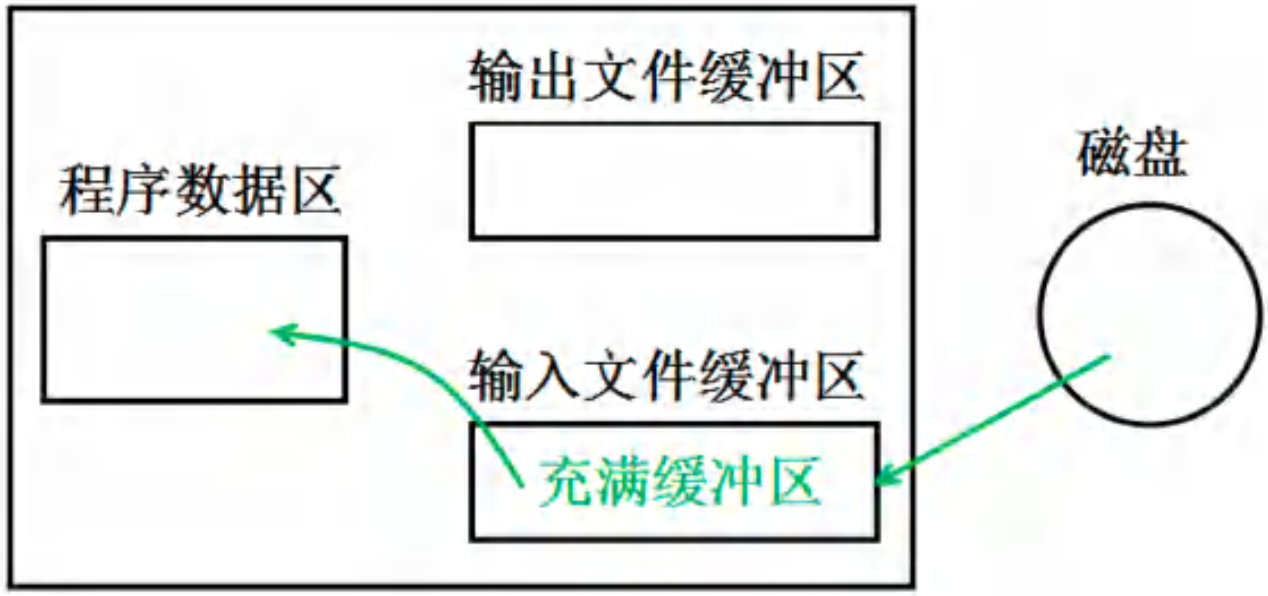
ANSI C标准采用“缓冲文件系统”处理数据文件 所谓缓冲文件系统是指系统自动地在内存区为程序中每一个正在使用的文件开辟一个文件缓冲区 从内存向磁盘输出数据必须先送到内存中的缓冲区,装满缓冲区后才一起送到磁盘去如果从磁盘向计算机读入数据,则一次从磁盘文件将一批数据输入到内存缓冲区(充满缓冲 区),然后再从缓冲区逐个地将数据送到程序数据区(给程序变量)

2)输入输出流

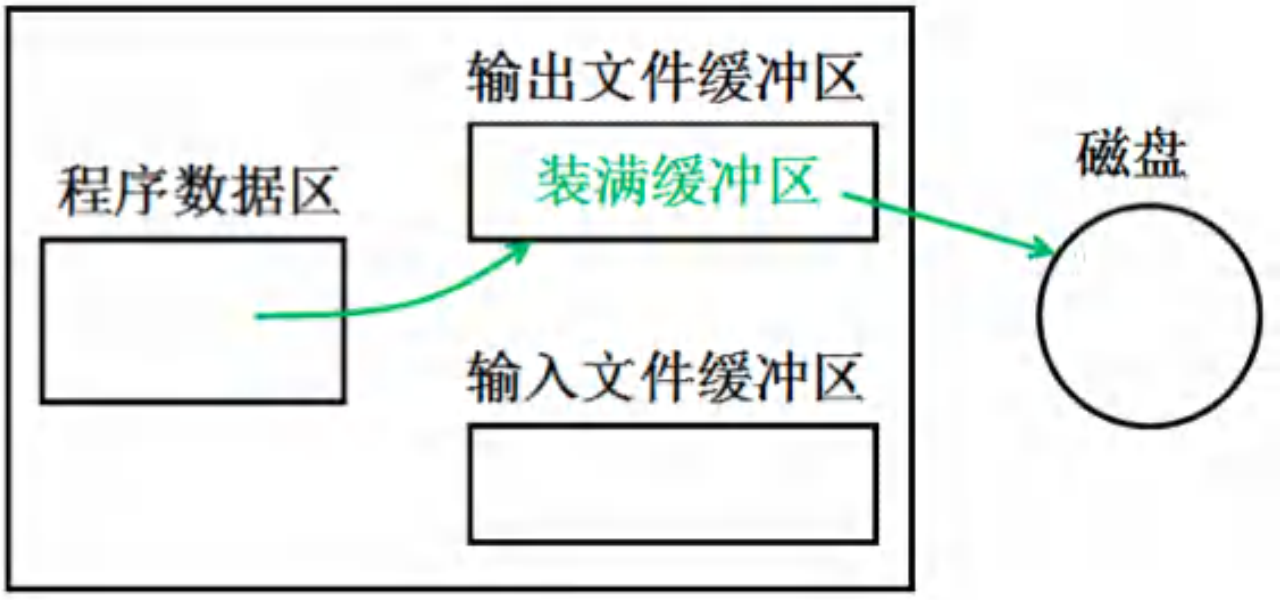
输入输出是数据传送的过程,数据如流水一样从一处流向另一处,因此常将输入输出形象地称为 流(stream),即数据流。流表示了信息从源到目的端的流动。



输入操作时,数据从文件流向计算机内存 --- 文件的读取



输出操作时,数据从计算机流向文件 --- 文件的写入



无论是用Word打开或保存文件,还是C程序中的输入输出都是通过操作系统进行的 “流” 是一个传输通道,数据可以从运行环境流入程序中,或从程序流至运行环境

- 文件句柄

```
typedef struct
{
    short           level;      /* 缓冲区"满"或者"空"的程度 */
    unsigned        flags;      /* 文件状态标志 */
    char            fd;         /* 文件描述符 */
    unsigned char   hold;       /* 如无缓冲区不读取字符 */
    short           bsize;      /* 缓冲区的大小 */
    unsigned char   *buffer;    /* 数据缓冲区的位置 */
    unsigned        ar;         /* 指针，当前的指向 */
    unsigned        istemp;     /* 临时文件，指示器 */
    short           token;      /* 用于有效性的检查 */
}FILE;
```

9.3 C语言文件指针

在C语言中用一个指针变量指向一个文件,这个指针称为文件指针。

声明FILE结构体类型的信息包含在头文件“stdio.h”中
一般设置一个指向FILE类型变量的指针变量,然后通过它来引用这些FILE类型变量 通过文件指针就可对它所指的文件进行各种操作。
定义说明文件指针的一般形式为:

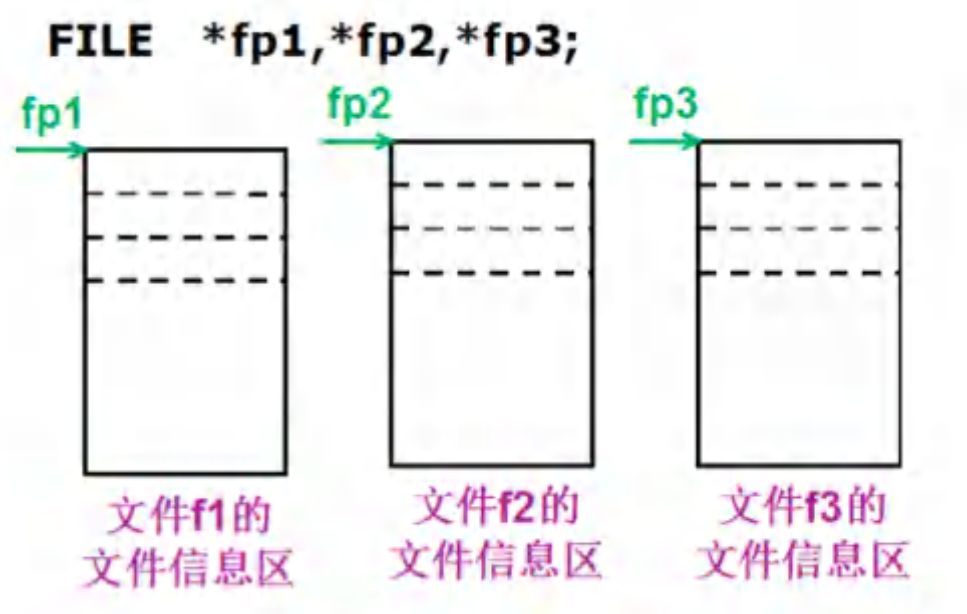
```
FILE * 指针变量标识符;
```

其中FILE应为大写,它实际上是由系统定义的一个结构,该结构中含有文件名、文件状态和文件 当前位置等信息。在编写源程序时不必关心FILE结构的细节。

```
FILE *fp;
```

表示fp是指向FILE结构的指针变量,通过fp即可找存放某个文件信息的结构变量,然后按结构变 量供的信息找到该文件,实施对文件的操作。习惯上也笼

统地把fp称为指向一个文件的指针。



9.4 文件操作API

fgetc	fputc	按照字符读写文件
fputs	fgets	按照行读写文件 （读写配置文件）
fread	fwirte	按照块读写文件 （大数据块迁移）
fprintf		按照格式化进行读写文件

9.5 标准的文件读写

1. 文件的打开fopen()

文件的打开操作表示将给用户指定的文件在内存分配一个FILE结构区，并将该结构的指针返回给用户程序，以后用户程序就可用此FILE指针来实现对指

定文件的存取操作了。当使用打开函数时，必须给出文件名、文件操作方式(读、写或读写),如果该文件名不存在，就意味着建立(只对写文件而言，对读文件则出错)，并将文件指针指向文件开头。若已有一个同名文件存在，则删除该文件，若无同名文件，则建立该文件，并将文件指针指向文件开头。

```
fopen(char *filename,char *type);
```

其中*filename是要打开文件的文件名指针，一般用双引号括起来的文件

方式	含义
“r”	打开，只读，文件必须已经存在。
“w”	只写,如果文件不存在则创建,如果文件已存在则把文件长度截断(Truncate)为0字节。再重新写,也就是替换掉原来的文件内容文件指针指到头。
“a”	只能在文件末尾追加数据,如果文件不存在则创建
“rb”	打开一个二进制文件，只读
“wb”	打开一个二进制文件，只写
“ab"	打开一个二进制文件，追加
“r+”	允许读和写,文件必须已存在
“w+”	允许读和写,如果文件不存在则创建,如果文件已存在则把文件长度截断为0字节再重新写 。
“a+”	允许读和追加数据,如果文件不存在则创建
“rb+”	以读/写方式打开一个二进制文件
“wb+”	以读/写方式建立一个新的二进制文件
“ab+”	以读/写方式打开一个二进制文件进行追加

名表示，也可使用双反斜杠隔开的路径名。而*type参数表示了对打开文件的操作方式。其可采用的操作方式如下：

当用fopen()成功的打开一个文件时，该函数将返回一个FILE指针，如果文件打开失败，将返回一个NULL指针。如想打开test文件，进行写：

```
FILE *fp;

if((fp=fopen("test","w"))==NULL)
{
    printf("File cannot be    opened\n");
    exit();
}
else
{
    printf("File opened for writing\n");
}

fclose(fp);
```

2. 关闭文件函数fclose()

文件操作完成后，必须要用fclose()函数进行关闭，这是因为对打开的文件进行写入时，若文件缓冲区的空间未被写入的内容填满，这些内容不会写到打开的文件中去而丢失。只有对打开的文件进行关闭操作时，停留在文件缓冲区的内容才能写到该文件中去，从而使文件完整。再者一旦关闭了文件，该文件对应的FILE结构将被释放，从而使关闭的文件得到保护，因为这时对该文件的存取操作将不会进行。文件的关闭也意味着释放了该文件的缓冲区。

```
int fclose(FILE *stream);
```

它表示该函数将关闭FILE指针对应的文件，并返回一个整数值。若成功地关闭了文件，则返回一个0值，否则返回一个非0值。常用以下方法进行测试

```
if(fclose(fp)!=0)
{
    printf("File cannot be closed\n");
    exit(1);
}
else
{
    printf("File is now closed\n");
}
```

3.文件的读写

(1).读写文件中字符的函数(一次只读写文件中的一个字符):

```
int fgetc(FILE *stream);
int fputc(int ch,FILE *stream);

int getc(FILE *stream);
int putc(int ch,FILE *stream);
```

其中fgetc()函数将把由流指针指向的文件中的一个字符读出，例如：

```
   ch=fgetc(fp);
```

将把流指针fp指向的文件中的一个字符读出，并赋给ch，当执行fgetc()函数时，若当时文件指针指到文件尾，即遇到文件结束标志EOF(其对应值为-1)，该函数返回一个-1给ch，在程序中常用检查该函数返回值是否为-1来判断是否已读到文件尾，从而决定是否继续。

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char ch;

    if((fp=fopen("myfile.txt","r"))==NULL)
    {
        printf("file cannot be opened\n");
        exit(1);
    }
    while((ch=fgetc(fp))!=EOF) {
        fputc(ch,stdout);
    }

    fclose(fp);

    return 0;
}
```

该程序以只读方式打开myfile.txt文件，在执行while循环时，文件指针每循环一次后移一个字符位置。用fgetc()函数将文件指针指定的字符读到ch变量中，然后用fputc()函数在屏幕上显示，当读到文件结束标志EOF时，变关闭该文件。

上面的程序用到了fputc()函数，该函数将字符变量ch的值写到流指针指定的文件中去，由于流指针用的是标准输出(显示器)的FILE指针stdout，故读出的字符将在显示器上显示。

(2).读写文件中字符串的函数

```
char *fgets(char *string,int n,FILE *stream);
int fprintf(FILE *stream,char *format, ...);
int fputs(char *string,FILE *stream);
```

其中fgets()函数将把由流指针指定的文件中n-1个字符，读到由指针stream指向的字符数组中去，例如：

```
fgets(buffer,9,fp);
```

将把fp指向的文件中的8个字符读到buffer内存区，buffer可以是定义的字符数组，也可以是动态分配的内存区。

注意，fgets()函数读到'\n'就停止，而不管是否达到数目要求。同时在读取字符串的最后加上'\0'。

fgets()函数执行完以后，返回一个指向该串的指针。如果读到文件尾或出错，则均返回一个空指针NULL，所以长用feof()函数来测定是否到了文件尾或者是ferror()函数来测试是否出错，例如下面的程序用fgets()函数读test.txt文件中的第一行并显示出来：

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char str[128];

    if((fp=fopen("test.txt","r"))==NULL)
    {
        printf("cannot open file\n");
        exit(1);
    }

    while(!feof(fp))
    {
        if(fgets(str,128,fp)!=NULL) printf("%s",str);
    }

    fclose(fp);
```

```
    return 0;
}
```

fputs()函数想指定文件写入一个由string指向的字符串，'\0'不写入文件。

fprintf()同printf()函数类似，不同之处就是printf()函数是想显示器输出，fprintf()则是向流指针指向的文件输出。

下面程序是向文件test.dat里输入一些字符：

```
#include<stdio.h>

int main(void)
{
    char *s="That's good news";
    int i=617;
    FILE *fp;

    fp=fopen("test.dat", "w");

    fputs("Your score of TOEFLis",fp);

    fputc(':', fp);
    fprintf(fp, "%d\n", i);
    fprintf(fp, "%s", s);

    fclose(fp);

    return 0;
}
```

文件中内容：

Your score of TOEFL is: 617
That's good news

4.清除和设置文件缓冲区

```
int fflush(FILE *stream);
```

fflush()函数将清除由stream指向的文件缓冲区里的内容，常用于写完一些数据后，立即用该函数清除缓冲区，以免误操作时，破坏原来的数据。

5.文件的随机读写函数

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
//返回值:成功返回0,出错返回-1并设置errno

long ftell(FILE *stream);
//返回值:成功返回当前读写位置,出错返回-1并设置errno

void rewind(FILE *stream);
```

fseek的whence和offset参数共同决定了读写位置移动到何处,whence参数的含义如下:

SEEK_SET
从文件开头移动offset个字节
SEEK_CUR
从当前位置移动offset个字节
SEEK_END
从文件末尾移动offset个字节

offset可正可负,负值表示向前(向文件开头的方向)移动,正值表示向后(向文件末尾的方 向)移动,如果向前移动的字节数超过了文件开头则出错返回,如果向后移动的字节数超过了 文件末尾,再次写入时将增大文件尺寸,从原来的文件末尾到fseek移动之后的读写位置之间的 字节都是0。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE* fp;

    if ( (fp = fopen("textfile","r+")) == NULL) {
        printf("Open file textfile\n”);
        exit(1);
    }

    if (fseek(fp, 10, SEEK_SET) != 0) {
        printf("Seek file textfile\n");
        exit(1);
    }

    fputc('K', fp);
```

```
    fclose(fp);

    return 0;
}
```

(2). 文件二进制块读写函数

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

//返回值:读或写的记录数,成功时返回的记录数等于nmemb,
//出错或读到文件末尾时返回 的记录数小于nmemb,也可能返回0
```

fread和fwrite用于读写记录,这里的记录是指一串固定长度的字节,比如一个int、一个结构 体或者一个定长数组。参数size指出一条记录的长度,而nmemb指出要读或写多少条记录,这些 记录在ptr所指的内存空间中连续存放,共占size * nmemb个字节,fread从文件stream中读 出size * nmemb个字节保存到ptr中,而fwrite把ptr中的size * nmemb个字节写到文件stream中。

nmemb是请求读或写的记录数,fread和fwrite返回的记录数有可能小于nmemb指定的记录数。例 如当前读写位置距文件末尾只有一条记录的长度,调用fread时指定nmemb为2,则返回值为1。 如果当前读写位置已经在文件末尾了,或者读文件时出错了,则fread返回0。如果写文件时出 错了,则fwrite的返回值小于nmemb指定的值。下面的例子由两个程序组成,一个程序把结构体 保存到文件中,另一个程序和从文件中读出结构体。

```
#include <stdio.h>
#include <stdlib.h>

struct record {
    char name[10];
    int age;
};

int main(void)
{
    struct record array[2] = {{"Ken", 24}, {"Knuth", 28}};

    FILE *fp = fopen("recfile", "w");
```

```
    if (fp == NULL) {
        printf("Open file recfile");
        exit(1);
    }

    fwrite(array, sizeof(struct record), 2, fp); fclose(fp);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>

struct record {
    char name[10];
    int age;
};

int main(void)
{
    struct record array[2];

    FILE *fp = fopen("recfile", "r");

    if (fp == NULL) {
        printf("Open file recfile");
        exit(1);
    }

    fread(array, sizeof(struct record), 2, fp);
    printf("Name1: %s\tAge1: %d\n", array[0].name, array[0].age);
    rintf("Name2: %s\tAge2: %d\n", array[1].name, array[1].age);

    fclose(fp);

    return 0;
}
```

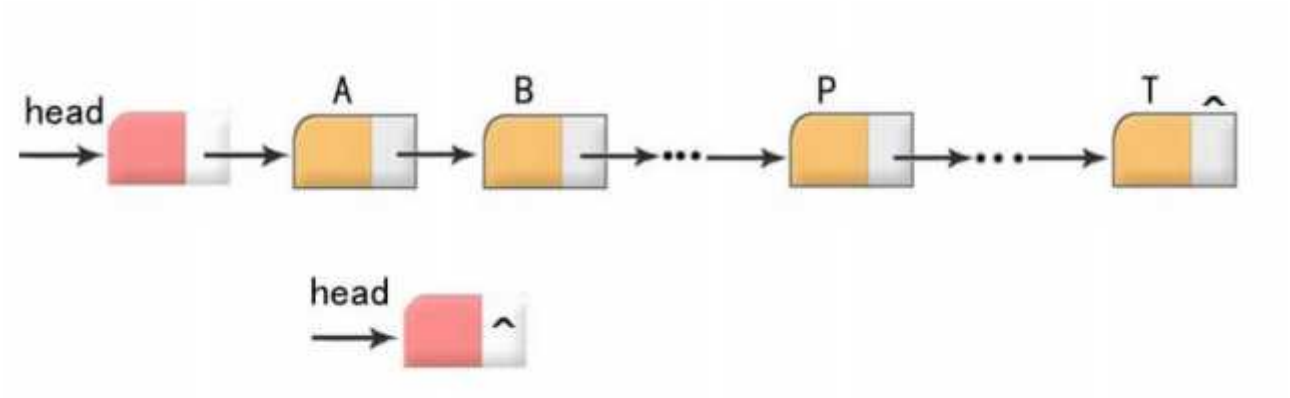
9.6 文件操作案例

（1）配置文件读写

配置文件读写案例实现分析

1、 功能划分
   a) 界面测试（功能集成）
      自己动手规划接口模型。
   b) 配置文件读写
      i. 配置文件读（根据key，读取valude）
      ii. 配置文件写（输入key、valude）
      iii. 配置文件修改（输入key、valude）
      iv. 优化 ===》接口要求紧 模块要求松
2、 实现及代码讲解
3、 测试。

10.  天使链表



- 链表是一种常用的数据结构，它通过指针将一些列数据结点，连接成一个数据链。相对于数组，链表具有更好的动态性（非顺序存储）。
- 数据域用来存储数据，指针域用于建立与下一个结点的联系。
- 建立链表时无需预先知道数据总量的，可以随机的分配空间，可以高效的在链表中的任意位置实时插入或删除数据。
- 链表的开销，主要是访问顺序性和组织链的空间损失。

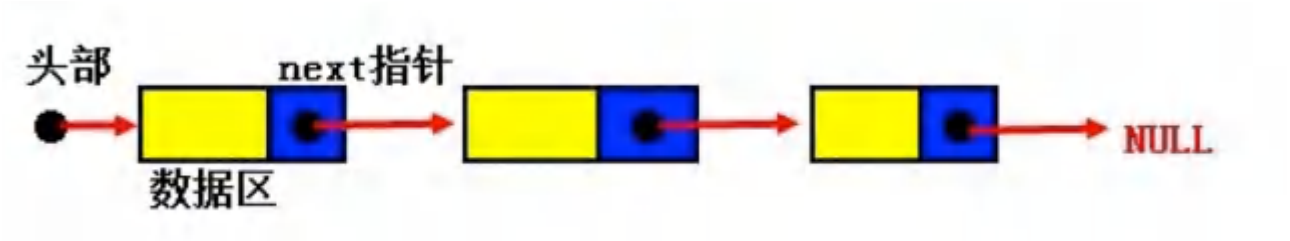
10.1 链表的相关概念

1）有关结构体的自身引用

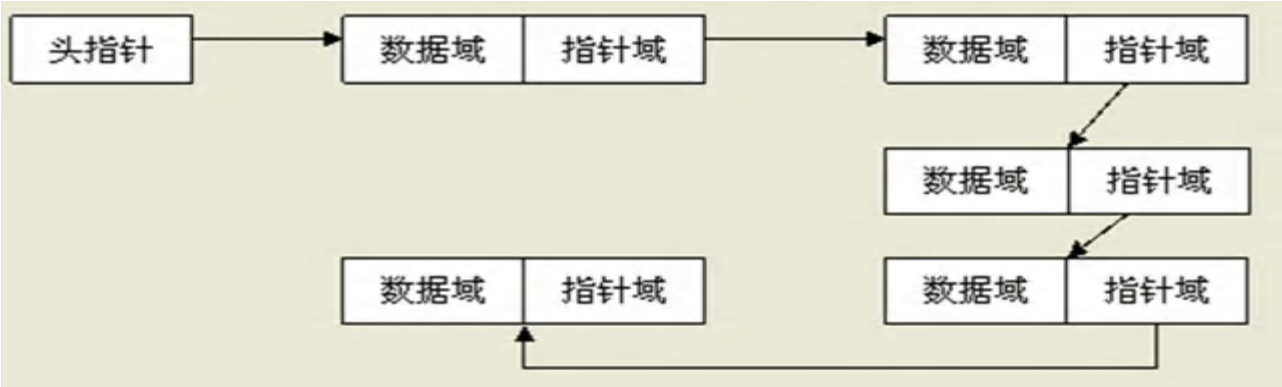
```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//结构体嵌套结构体指针(√)
typedef struct Teacher
{
    char name[64];
    int id;
```

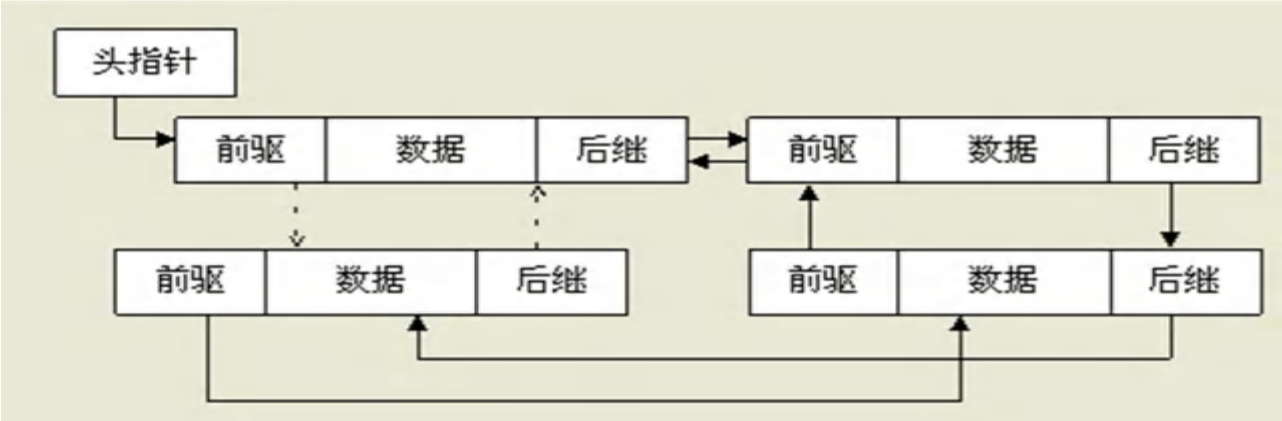
```
    struct Teacher *teacher;
} teacher_t;


//数据类型本质：固定大小内存块的别名

//结构体中套一个结构体(X)
typedef struct Student
{
    char name[64];
    int id;
    struct Student student;
} student_t;
//在自己类型大小 还没有确定的情况下 引用自己类型的元素 是不正确的
//结构体不能嵌套定义 （确定不了数据类型的内存大小，分配不了内存）


int main(void)
{
    teacher_t t1;
    student_t s1;

    return 0;
}
```

2） data域和指针域




```
typedef struct node *link;

struct node {
    unsigned char item;      //data域
    link next;               //链表域
};
```





10.2 分类

1）动态链表和静态链表

静态链表和动态链表是线性表链式存储结构的两种不同的表示方式。

静态链表的初始长度一般是固定的，在做插入和删除操作时不需要移动元素，仅需修改指针，故仍具有链式存储结构的主要优点。

动态链表是相对于静态链表而言的，一般地，在描述线性表的链式存储结构时如果没有特别说明即默认描述的是动态链表。

静态链表

```c
#include <stdio.h>
#include <stdlib.h>

/*所有结点都是在程序中定义的，不是临时开辟的，也不能用完后释放，这种链表称为“静态链表”。*/

struct Student
{
    int num;
    float score;
    struct Student *next;
};

int main(void)
{
    struct Student stu1, stu2, stu3, *head, *p;

    stu1.num = 1001; stu1.score = 80; //对结点stu1的num和score成员赋值
    stu2.num = 1002; stu2.score = 85; //对结点stu2的num和score成员赋值
    stu3.num = 1003; stu3.score = 90; //对结点stu3的num和score成员赋值

    head = &stu1;       //头指针指向第1个结点stu1
    stu1.next = &stu2; //将结点stu2的地址赋值给stu1结点的next成员
    stu2.next = &stu3; //将结点stu3的地址赋值给stu2结点的next成员
    stu3.next = NULL;  //stu3是最后一个结点，其next成员不存放任何结点的地址，置为NULL
    p = head;           //使p指针也指向第1个结点

    //遍历静态链表
    do{
        printf("%d,%f\n", p->num, p->score); //输出p所指向结点的数据
        p = p->next;                          //然后让p指向下一个结点
    } while (p != NULL);                      //直到p的next成员为NULL，即完成遍历


    return 0;
}
```

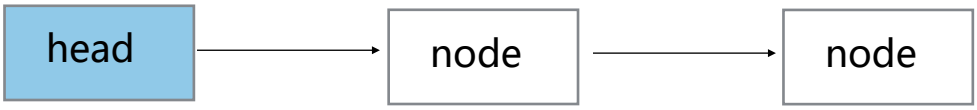
动态链表

```c
#include <stdio.h>
#include <stdlib.h>

/*所谓动态链表，是指在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点和输入各结点数据，并建立起前后相链的关系。*/

struct Student
{
    int No;//学号
    struct Student *next;
};

int main(void)
{
    struct Student *p1, *p2, *head;
    struct Student *p;
    int n = 0; //结点个数

    head = NULL;
    p1 = (struct Student *)malloc(sizeof(struct Student));
    printf("请输入1个学号\n");
    scanf("%d", &p1->No);
    p2 = p1; //开始时，p1和p2均指向第1个结点

    while (p1->No != 0)
    {
        n++;
        if (n == 1)
        {
            head = p1;
        }
        else
        {
            p2->next = p1;
        }
        p2 = p1;//p2是最后一个结点
        printf("请输入学号，输入0终止：\n");
        p1 = (struct Student *)malloc(sizeof(struct Student));
        scanf("%d", &p1->No);
    };
    p2->next = NULL;//输入完毕后，p2->next为NULL

    //遍历动态链表
    p = head;
    while (p != NULL)
    {
        printf("%d,", p->No);
        p = p -> next;
    }
    printf("\n");
```

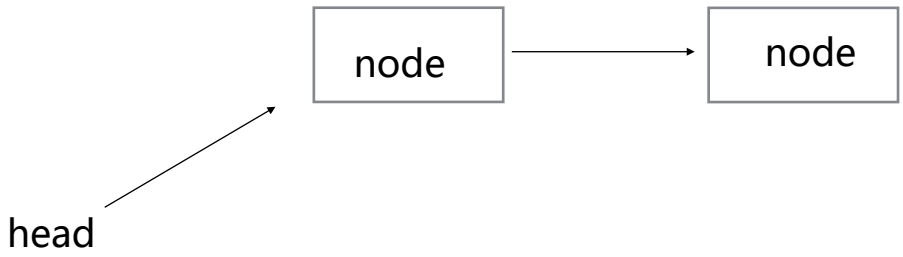
```
    return 0;
}
```

2）带头链表和不带头链表

```
typedef struct node *link;

struct node {
    unsigned char item;     //data域
    link next;              //链表域
};


link head;      //声明头结点
```




```
 /* 带头结点初始化 */
void list_init(link *head){
    *head=(link)malloc(sizeof(struct node));
    (*head)->next=NULL;
}
```




```
/* 不带头结点初始化 */
void list_init(link *head){
      *head=NULL;
}
```

10.3 基本操作

（1）单向链表（无头结点例子）

```
/* linkedlist.h */

#ifndef LINKEDLIST_H
#define LINKEDLIST_H

typedef struct node *link;

struct node {
    unsigned char item;
    link next;
};

link list_init(link *head_p);

/* 创建一个链表节点 */
link make_node(unsigned char item);

/* 释放一个链表节点 */
void free_node(link p);

/* 查找一个链表*/
link search(link head, unsigned char key);

/* 插入一个链表 */
void insert(link *head, link p);

/* 删除一个链表 */
link delete(link *head, link p);

/* 遍历一个链表 */
void traverse(link head, void (*visit)(link) );

/* 销毁所有链表 */
void destory(link *head);

#endif
```

```
/* linkedlist.c */

#include <stdlib.h>
```

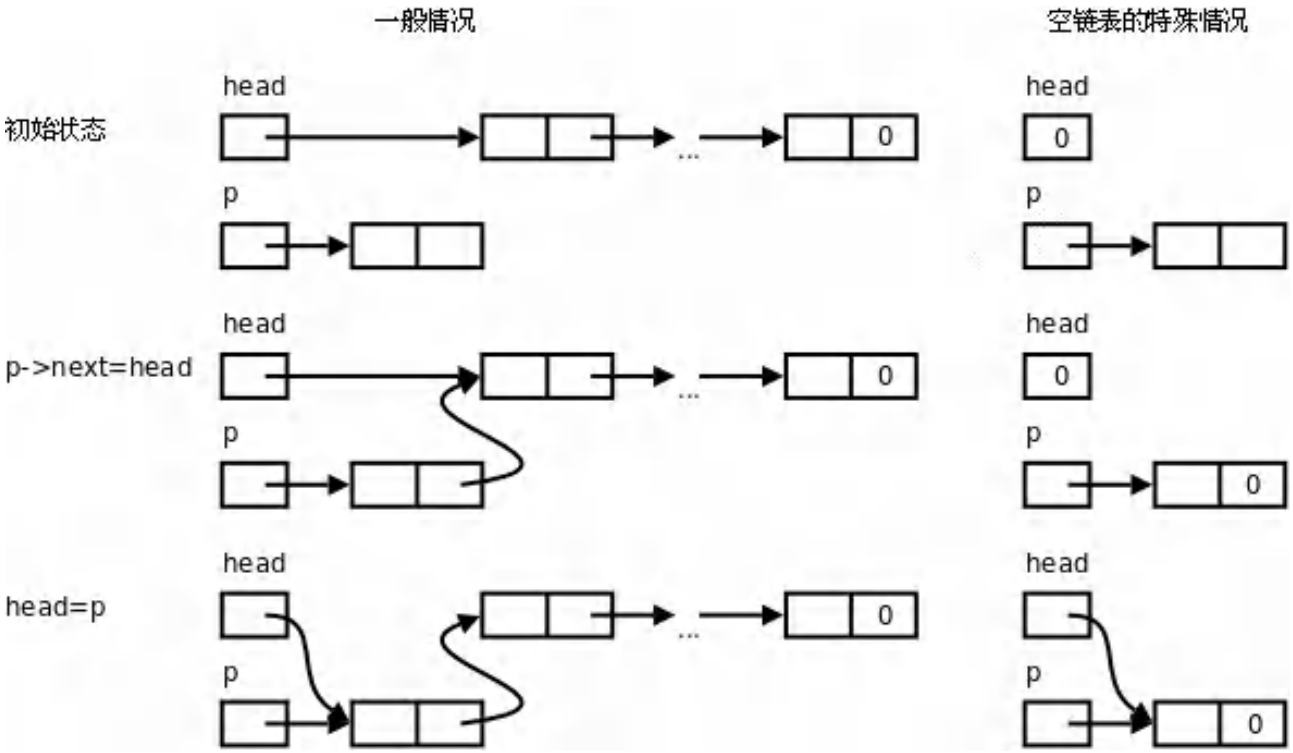
```c
#include <stdio.h>
#include "linkedlist.h"


link list_init(link *head_p)
{
   *head_p = NULL;

   return *head_p;
}

link make_node(unsigned char item)
{
    link p = malloc(sizeof *p);
    p->item = item;
    p->next = NULL;

    return p;
}

void free_node(link p)
{
    free(p);
}

link search(link head, unsigned char key)
{
    link p;
    for (p = head; p; p = p->next) {
        if (p->item == key) {
            return p;
        }
    }

    return NULL;
}

void insert(link *head, link p)
{
    p->next = *head;
    *head = p;
}

link delete(link *head, link p)
{
    link prev;

    if (p == *head) {
        *head = p->next;
        return p;
    }

    for (prev = *head; prev; prev = prev->next) {
        if (prev->next == p) {
            prev->next = p->next;
```

```
            return p;
        }
    }

    return NULL;
}

void traverse(link head, void (*visit)(link) ) {
    link p;

    for (p = head; p; p = p->next) {
        visit(p);
    }
}

void destory(link *head)
{
    link q = NULL, p = *head;

    while (p) {
        q = p;
        p = p->next;
        free(q);
    }

    *head = NULL;
}

void print_item(link p)
{
    printf("%d\n", p->item);
}

int main(void)
{
    link head;

    list_init(&head);

    link p = make_node(10);
    insert(&head, p);
    p = make_node(90);
    insert(&head, p);
    p = search(head, 10);
    delete(&head, p);
    free_node(p);

    p = make_node(30);
    insert(&head, p);
    traverse(head, print_item);

    destory(&head);

    return 0;
}
```

在初始化时把头指针head初始化为NULL,表示空链表。然后main函数调用make_node创建几个节 点,分别调用insert插入到链表中。

链表的插入操作

```
void insert(link *head, link p)
{
    p->next = *head;
    *head = p;
}
```



正如上图所示,insert函数虽然简单,其中也隐含了一种特殊情况(Special Case)的处理, 当head为NULL时,执行insert操作插入第一个节点之后,head指向第一个节点,而第一个节点的next指针域成为NULL,这很合理,因为它也是最后一个节点。所以空链表虽然是一种特殊情 况,却不需要特殊的代码来处理,和一般情况用同样的代码处理即可,这样写出来的代码更简 洁,但是在读代码时要想到可能存在的特殊情况。当然,insert函数传进来的参数p也可能有特 殊情况,传进来的p可能是NULL,甚至是野指针,本章的函数代码都假定调用者的传进来的参数 是合法的,不对参数做特别检查。事实上,对指针参数做检查是不现实的,如果传进来的 是NULL还可以检查一下,如果传进来的是野指针,根本无法检查它指

向的内存单元是不是合法 的,C标准库的函数通常也不做这种检查,例如strcpy(p, NULL)就会引起段错误。

接下来main函数调用search在链表中查找某个节点,如果找到就返回指向该节点的指针,找不到就返回NULL。

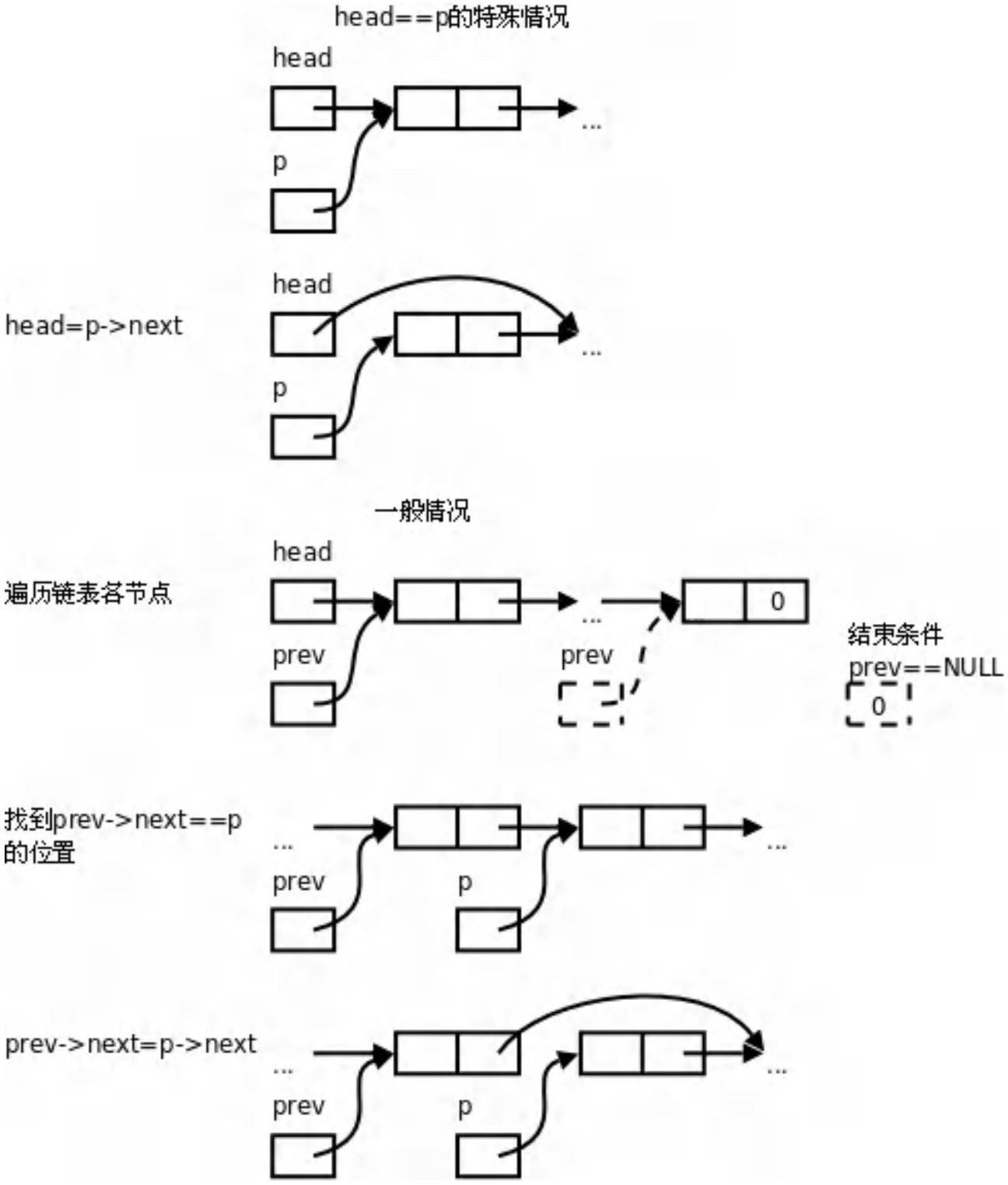
```
link search(link head, unsigned char key)
{
    link p;
    for (p = head; p; p = p->next) {
        if (p->item == key) {
            return p;
        }
    }

    return NULL;
}
```

search函数其实也隐含了对于空链表这种特殊情况的处理,如果是空链表则循环体一次都不执 行,直接返回NULL。

然后main函数调用delete从链表中摘除用search找到的节点,最后调用free_node释放它的存储 空间。

链表的删除操作

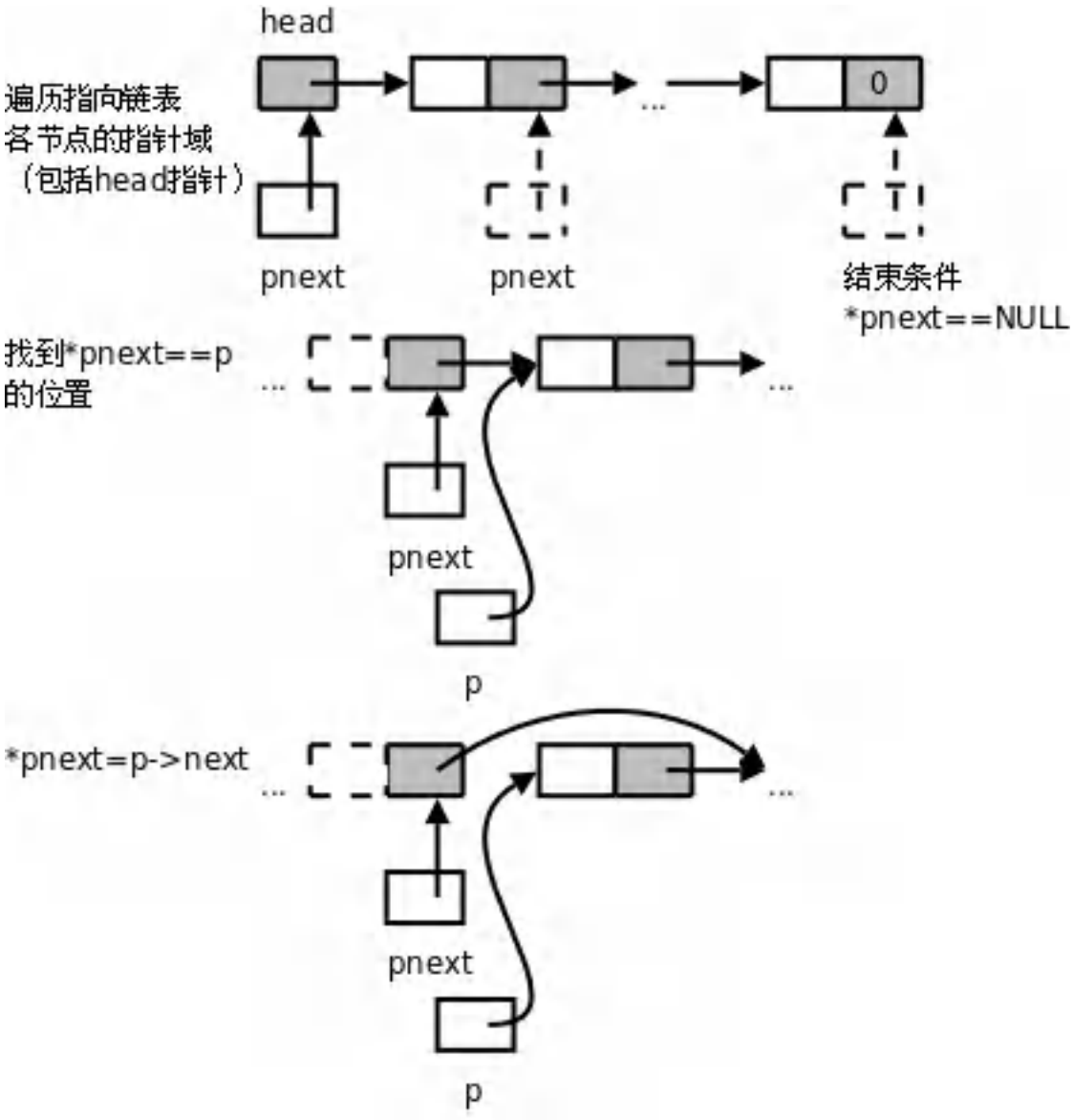
```
link delete(link *head, link p)
{
    link prev;

    if (p == *head) {
        *head = p->next;
        return p;
    }

    for (prev = *head; prev; prev = prev->next) {
        if (prev->next == p) {
            prev->next = p->next;
            return p;
        }
    }

    return NULL;
}
```



从上图可以看出,要摘除一个节点需要首先找到它的前趋然后才能做摘除操作,而在单链表中 通过某个节点只能找到它的后继而不能找到它的前趋,所以删除操作要麻烦一些,需要从第一 个节点开始依次查找要摘除的节点的前趋。delete操作也要处理一种特殊情况,如果要摘除的 节点是链表的第一个节点,它是没有前趋的,这种情况要用特殊的代码处理,而不能和一般情 况用同样的代码

处理。这样很不爽,能不能把这种特殊情况转化为一般情况呢?可以 把delete函数改成这样:

```
link delete(link *head, link p)
{
    link *pnext;

    for (pnext = head; *pnext; pnext = &(*pnext)->next) {
        if (*pnext == p) {
            *pnext = p->next;
            return p;
        }
    }

    return NULL;
}
```



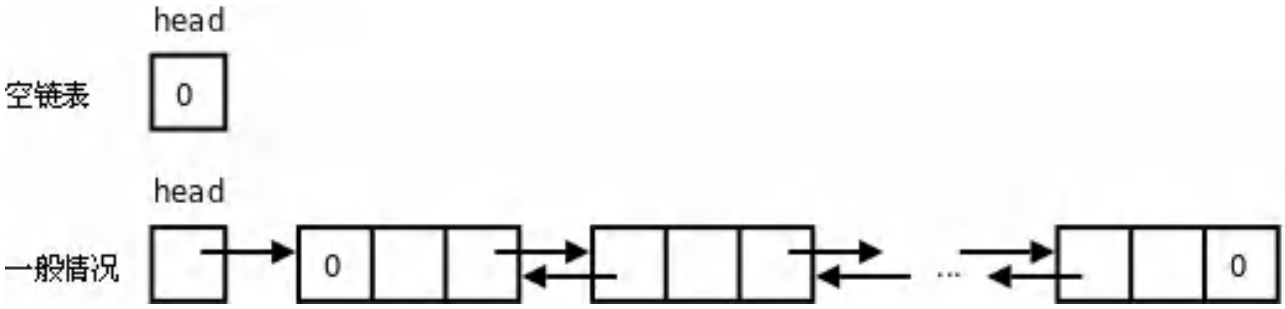
定义一个指向指针的指针pnext,在for循环中pnext遍历的是指向链表中各节点的指针域,这样 就把head指针和各节点的next指针统一起来了,可以在一个循环中处理。

然后main函数调用traverse函数遍历整个链表,调用destroy函数销毁整个链表。请读者自己阅 读这两个函数的代码。

(2) 双向链表（带有头结点例子）

```
struct node {
    unsigned char item;
    link prev, next;
};
```

无头节点结构



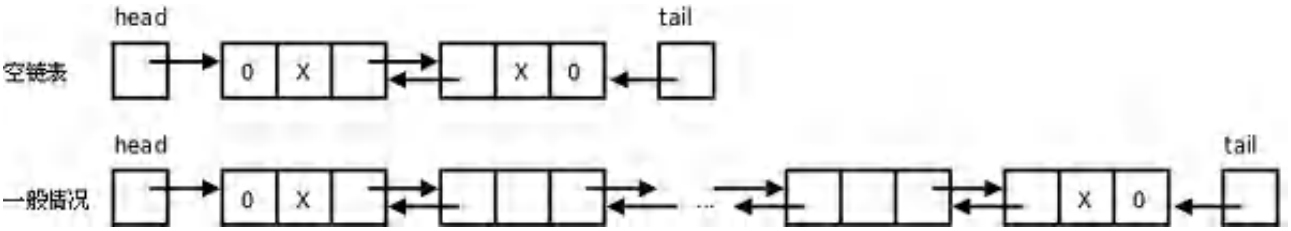
链表的delete操作需要首先找到要摘除的节点的前趋,而在单链表中找某个节点的前趋需要从表头开始依次查找,对于n个节点的链表,删除操作的时间复杂度为O(n)。可以想像得到,如果每个节点再维护一个指向前趋的指针,删除操作就像插入操作一样容易了,时间复杂度 为 O(1),这称为双向链表(Doubly Linked List)。要实现双向链表只需在上一节代码的基础上 改动两个地方。

```
void insert(link p)
{
    p->next = head;

    if (head) {
        head->prev = p;
    }

    head = p;
    p->prev = NULL;
```

```
}

link delete(link p)
{
    if (p->prev) {
        p->prev->next = p->next;
    }
    else {
        head = p->next;
    }
    if (p->next) {
        p->next->prev = p->prev;
    }

    return p;
}
```

由于引入了prev指针,insert和delete函数中都有一些特殊情况需要用特殊的代码处理,不能和 一般情况用同样的代码处理,这非常不爽,如果在表头和表尾各添加一个Sentinel节点(这两个节点只用于界定表头和表尾,不保存数据),就可以把这些特殊情况都转化为一般情况了。

有头节点结构




```
/* doublylinkedlist.h */

#ifndef DOUBLYLINKEDLIST_H
#define DOUBLYLINKEDLIST_H

typedef struct node *link;
struct node {
    unsigned char item;
    link prev, next;
};

/* 创建节点 */
link make_node(unsigned char item);
```

```c
/* 释放节点 */
void free_node(link p);

/* 插入节点 */
void insert(link p);

/* 删除节点 */
link delete(link p);

/* 查找节点 */
link search(unsigned char key);

/* 遍历节点 */
void traverse(void (*visit)(link));

/* 销毁链表 */
void destroy(void);

#endif
```

```c
/* doublylinkedlist.c */

#include <stdio.h>
#include <stdlib.h>
#include "doublelist.h"

struct node tailsentinel;
struct node headsentinel = {0, NULL, &tailsentinel};
struct node tailsentinel = {0, &headsentinel, NULL};

static link head = &headsentinel;
static link tail = &tailsentinel;

link make_node(unsigned char item)
{
    link p = malloc(sizeof *p);
    p->item = item;
    p->prev = p->next = NULL;
    return p;
}
void free_node(link p)
{
    free(p);
}

link search(unsigned char key)
{
    link p;
    for (p = head->next; p != tail; p = p->next) {
        if (p->item == key)  {
            return p;
        }
```

```
    }
    return NULL;
}

void insert(link p)
{
    p->next = head->next;
    head->next->prev = p;
    head->next = p;
    p->prev = head;
}

link delete(link p)
{
    p->prev->next = p->next;
    p->next->prev = p->prev;
    return p;
}

void traverse(void (*visit)(link))
{
    link p;
    for (p = head->next; p != tail; p = p->next) {
        visit(p);
    }
}

void destroy(void)
{
    link q, p = head->next;

    head->next = tail;
    tail->prev = head;

    while (p != tail) {
        q = p;
        p = p->next;
        free(q);
    }
}


void print_item(link p) {
    printf("%d\n", p->item);
}

int main(void)
{
    link p = make_node(10);
    insert(p);

    p = make_node(5);
    insert(p);

    p = make_node(90);
```

```
    insert(p);

    p = search(5);
    delete(p);
    free_node(p);

    traverse(print_item);

    destroy();

    return 0;
}
```

10.4 回调函数

```
void traverse(void (*visit)(link))
{
    link p;
    for (p = head->next; p != tail; p = p->next) {
        visit(p);
    }
}
```

```
void (*visit)(link)
```

是一个函数类型的形式参数，代表是一个参数，返回值是void，有一个参数是link的这一类的函数。

通常区分一个函数的类别，是用返回值和形参类型和个数来区分的。

(1) 使用回调函数步骤

1.写一个函数A,A里面有一个参数是个函数指针:

```
int funcA(int a, int (*Pcall)(int b));
//注意回调函数做形式参数函数名和*号要用括号搞在一起，否则 就会被返回值类型占有.
```

2.有个实体函数，那他要指向一个函数B,这个函数的类型应该和A的函数参数类型一样:

```
int funcB(int c);
```

3.使用A函数把参数赋值后,A中的形参Pcall函数指针指向了一个函数funB的地址:

```
funcA(36,funcB);
```

```
#include <stdio.h>

int funcB(int b)
{
    printf("in funcB:%d\n", b);
}

int funcA(int a,int (*Pcall)(int b))
{
    int PA = 3;
    int PS = 4;
    (*Pcall)(a);    //调用回调函数,传参数a
    (*Pcall)(PS);   //调用回调函数,传参数PS
    (*Pcall)(PA);   //调用回调函数,传参数PA
}

int main(void)
{
    funcA(3, funcB);    //将funcB当做参数传递给funcA
    funcB(5, funcB);

    return 0;
}
```

(2) 函数类型的别名

```
#include <stdio.h>

typedef int (*pCall)(int b); //将 int (*)(int b)类型的指针 起别名 pCall
```

```
pCall pCallA;            //定义一个函数指针
pCall pCallB;            //定义一个函数指针

int funcB(int b)
{
    printf("in funcB:%d\n", b);
}

int funcA(int a, pCallA)
{
    int PA = 3;
    int PS = 4;
    (*PcallA)(a);    //调用回调函数,传参数a
    (*PcallA)(PS);   //调用回调函数,传参数PS
    (*PcallA)(PA);   //调用回调函数,传参数PA
}

int main(void)
{
    funcA(3, funcB);    //将funcB当做参数传递给funcA
    funcB(5, funcB);

    return 0;
}
```

(3) 如何确定一个函数类型

```
(*(void(*)())0)();
```

这样的表达式会令C程序员心惊胆战。但是，并不需要这样，因为他们可以在一个简单的规 则的帮助下很容易地构造它：以你使用的方式声明它。

每个C变量声明都具有两个部分：一个类型和一组具有特定格式的期望用来对该类型求值的表达式。最简单的表达式就是一个变量：

```
float f, g;
```

说明表达式f和g——在求值的时候——具有类型float。由于待求值的是表达式，因此可以自由地使用圆括号：

```
float ((f));
```

这表示((f))求值为float并且因此，通过推断，f也是一个float。

同样的逻辑用在函数和指针类型。
例如：

```
float ff();
```

表示表达式ff()是一个float，因此ff是一个返回一个float的函数。类似地，

```
float *pf;
```

表示*pf是一个float并且因此pf是一个指向一个float的指针。
这些形式的组合声明对表达式是一样的。因此，

```
float *g(), (*h)();
```

表示*g()和(*h)()都是float表达式。由于()比*绑定得更紧密，*g()和*(g())表示同样的东西：g是一个返回指float指针的函数，而h是一个指向返回float的函数的指针。

当我们知道如何声明一个给定类型的变量以后，就能够很容易地写出一个类型的模型（cast）：只要删除变量名和分号并将所有的东西包围在一对圆括号中即可。因此，由于

```
float *g();
```

声明g是一个返回float指针的函数，所以(float *())就是它的模型。

有了这些知识的武装，我们现在可以准备解决(*(void(*)())0)()了。 我们可以将它分为两个部分进行分析。首先，假设我们有一个变量fp，它包含了一个函数指针，并且我们希望调用fp所指向的函数。可以这样写：

```
(*fp)();
```

如果fp是一个指向函数的指针，则*fp就是函数本身，因此(*fp)()是调用它的一种方法。
(*fp)中的括号是必须的，否则这个表达式将会被分析为*(fp())。我们现在要找一个适当 的表达式来替换fp。

这个问题就是我们的第二步分析。如果C可以读入并理解类型，我们可以写：

```
(*0)();
```

但这样并不行，因为*运算符要求必须有一个指针作为他的操作数。另外，这个操作数必须是一个指向函数的指针，以保证*的结果可以被调用。因此，我们需要将0转换为一个可以描述“指向一个返回void的函数的指针”的类型。

如果fp是一个指向返回void的函数的指针，则(*fp)()是一个void值，并且它的声明将会是
这样的：

```
void (*fp)();
```

因此，我们需要写：

```
void (*fp)();
(*fp)();
```

一旦我们知道了如何声明该变量，我们也就知道了如何将一个常数转换为该类型：只要从变量的声明中去掉名字即可。因此，我们像下面这样将0转换为一个“指向返回void的函数的指针”：

```
(void(*)())0
```

接下来，我们用(void(*)())0来替换fp：

```
(*(void(*)())0)();
```

结尾处的分号用于将这个表达式转换为一个语句。

在这里，我们就解决了这个问题时没有使用typedef声明。通过使用它，我们可以更清晰地解决这个问题：

```
typedef void (*funcptr)();
(*(funcptr)0)();
```

✱ 奇葩的变量笔试题

给定以下类型的变量a的定义式：

a) 一个整型（An integer）
b) 一个指向整型的指针（A pointer to an integer）
c) 一个指向指向整型的指针（A pointer to a pointer to an integer）
d) 一个10个存放整型的数组（An array of 10 integers）
e) 一个10个存放指向整型指针的数组
（An array of 10 pointers to integers）
f) 一个指向存放10个整型数组的指针
（A pointer to an array of 10 integers）
g) 一个指向 需要一个整型参数并且返回值是一个整型函数的指针
（A pointer to a function that takes an integer as an argument and returns an integer）
h) 一个存放10个 指向 需要一个整型参数并且返回值是一个整型函数的指针的数组
（An array of ten pointers to functions that take an integer argument and return an integer）

11.  恶魔老巢动态库

```
/*
   下面定义了一套socket客户端发送报文接受报文的api接口
   请写出这套接口api的调用方法
 */

#ifndef _INC_Demo01_H
#define _INC_Demo01_H

#ifdef  __cplusplus
extern "C" {
#endif

    //------------------第一套api接口---Begin--------------------------------//
    //客户端初始化 获取handle上下
    int cltSocketInit(void **handle /*out*/);

    //客户端发报文
    int cltSocketSend(void *handle /*in*/, unsigned char *buf /*in*/,  int
buflen /*in*/);

    //客户端收报文
    int cltSocketRev(void *handle /*in*/, unsigned char *buf /*in*/, int
*buflen /*in out*/);

    //客户端释放资源
    int cltSocketDestory(void *handle/*in*/);
    //------------------第一套api接口---End----------------------------------//


#ifdef  __cplusplus
}
#endif
```

windows动态库生成的文件有

socketClient.lib socketClient.dll

windows 动态库是dll文件和lib文件组合。

当发现windows动态库没有lib文件生成时候，需要在动态库中每个函数头部添加

```
__declspec(dllexport)
```

头衔。

附录（A）

Win32环境下动态链接库(DLL)编程原理

比较大的应用程序都由很多模块组成，这些模块分别完成相对独立的功能，它们彼此协作来完成整个软件系统的工作。其中可能存在一些模块的功能较为通用，在构造其它软件系统时仍会被使用。在构造软件系统时，如果将所有模块的源代码都静态编译到整个应用程序EXE文件中，会产生一些问题：一个缺点是增加了应用程序的大小，它会占用更多的磁盘空间，程序运行时也会消耗较大的内存空间，造成系统资源的浪费；另一个缺点是，在编写大的EXE程序时，在每次修改重建时都必须调整编译所有源代码，增加了编译过程的复杂性，也不利于阶段性的单元测试。

Windows系统平台上提供了一种完全不同的较有效的编程和运行环境，你可以将独立的程序模块创建为较小的DLL(Dynamic Linkable Library)文件，并可对它们单独编译和测试。在运行时，只有当EXE程序确实要调用这些DLL模块的情况下，系统才会将它们装载到内存空间中。这种 方式不仅减少了EXE文件的大小和对内存空间的需求，而且使这些DLL模块可以同时被多个应用程序使用。Microsoft Windows自己就将一些主要的系统功能以DLL模块的形式实现。例如IE中的一些基本功能就是由DLL文件实现的，它可以被其它应用程序调用和集成。

一般来说，DLL是一种磁盘文件（通常带有DLL扩展名），它由全局数据、服务函数和资源组成，在运行时被系统加载到进程的虚拟空间中，成为调用进程的 一部分。如果与其它DLL之间没有冲突，该文件通常映射到进程虚拟空间的同一地址上。DLL模块中包含各种导出函数，用于向外界提供服务。Windows 在加载DLL模块时将进程函数调用与DLL文件的导出函数相匹配。

在Win32环境中，每个进程都复制了自己的读/写全局变量。如果想要与其它进程共享内存，必须使用内存映射文件或者声明一个共享数据段。DLL模块需要的堆栈内存都是从运行进程的堆栈中分配出来的。

DLL现在越来越容易编写。Win32已经大大简化了其编程模式，并有许多来自AppWizard和MFC类库的支持。

一、导出和导入函数的匹配

DLL文件中包含一个导出函数表。这些导出函数由它们的符号名和称为标识号的整数与外界联系起来。函数表中还包含了DLL中函数的地址。当应用程序加载 DLL模块时时，它并不知道调用函数的实际地址，但它知道函数的符号名和标识号。动态链接过程在加载的DLL模块时动态建立一个函数调用与函数地址的对应 表。如果重新编译和重建DLL文件，并不需要修改应用程序，除非你改变了导出函数的符号名和参数序列。

简单的DLL文件只为应用程序提供导出函数，比较复杂的DLL文件除了提供导出函数以外，还调用其它DLL文件中的函数。这样，一个特殊的DLL可以既有导入函数，又有导入函数。这并不是一个问题，因为动态链接过程可以处理交叉相关的情况。

在DLL代码中，必须像下面这样明确声明导出函数：

```
__declspec(dllexport) int MyFunction(int n);
```

但也可以在模块定义(DEF)文件中列出导出函数，不过这样做常常引起更多的麻烦。在应用程序方面，要求像下面这样明确声明相应的输入函数：

```
__declspec(dllimport) int MyFuncition(int n);
```

仅有导入和导出声明并不能使应用程序内部的函数调用链接到相应的DLL文件上。应用程序的项目必须为链接程序指定所需的输入库（LIB文件）。而且应用程序事实上必须至少包含一个对DLL函数的调用。

二、与DLL模块建立链接

应用程序导入函数与DLL文件中的导出函数进行链接有两种方式：隐式链接和显式链接。所谓的隐式链接是指在应用程序中不需指明DLL文件的实际存储路径，程序员不需关心DLL文件的实际装载。而显式链接与此相反。

采用隐式链接方式，程序员在建立一个DLL文件时，链接程序会自动生成一个与之对应的LIB导入文件。该文件包含了每一个DLL导出函数的符号名和可选 的标识号，但是并不含有实际的代码。LIB文件作为DLL的替代文件被编译到应用程序项目中。当程序员通过静态链接方式编译生成应用程序时，应用程

序中的 调用函数与LIB文件中导出符号相匹配，这些符号或标识号进入到生成的EXE文件中。LIB文件中也包含了对应的DLL文件名（但不是完全的路径名），链 接程序将其存储在EXE文件内部。当应用程序运行过程中需要加载DLL文件时，Windows根据这些信息发现并加载DLL，然后通过符号名或标识号实现对DLL函数的动态链接。

显式链接方式对于集成化的开发语言（例如VB）比较适合。有了显式链接，程序员就不必再使用导入文件，而是 直接调用Win32 的LoadLibary函数，并指定DLL的路径作为参数。LoadLibary返回HINSTANCE参数，应用程序在调用 GetProcAddress函数时使用这一参数。GetProcAddress函数将符号名或标识号转换为DLL内部的地址。假设有一个导出如下函数的 DLL文件：

```
extern "C" __declspec(dllexport) double SquareRoot(double d);
```

下面是应用程序对该导出函数的显式链接的例子：

```
typedef double(SQRTPROC)(double);
HINSTANCE hInstance;
SQRTPROC* pFunction;


VERIFY(hInstance=::LoadLibrary("c:\\winnt\\system32\\mydll.dll"));
VERIFY(pFunction=(SQRTPROC*)::GetProcAddress(hInstance,"SquareRoot"));


double d=(*pFunction)(81.0);//调用该DLL函数
```

在隐式链接方式中，所有被应用程序调用的DLL文件都会在应用程序EXE文件加载时被加载在到内存中；但如果采用显式链接方式，程序员可以决定DLL文 件何时加载或不加载。显式链接在运行时决定加载哪个DLL文件。例如，可以将一个带有字符串资源的DLL模块以英语加载，而另一个以西班牙语加载。应用程 序在用户选择了合适的语种后再加载与之对应的DLL文件。

三、使用符号名链接与标识号链接

在Win16环境中，符号名链接效率较低，所有那时标识号链接是主要的链接方式。在Win32环境中，符号名链接的效率得到了改善。Microsoft 现在推荐使用符号名链接。但在MFC库中的DLL版本仍然采用的是标识号链接。一个

典型的MFC程序可能会链接到数百个MFC DLL函数上。采用标识号链接的应用程序的EXE文件体相对较小，因为它不必包含导入函数的长字符串符号名。

四、编写DllMain函数

DllMain函数是DLL模块的默认入口点。当Windows加载 DLL模块时调用这一函数。系统首先调用全局对象的构造函数，然后调用全局函数DLLMain。DLLMain函数不仅在将DLL链接加载到进程时被调 用，在DLL模块与进程分离时（以及其它时候）也被调用。下面是一个框架DLLMain函数的例子。

```
HINSTANCE g_hInstance;

extern "C"
int APIENTRY DllMain(HINSTANCE hInstance,DWORD dwReason,LPVOID lpReserved)
{
    if(dwReason==DLL_PROCESS_ATTACH)
    {
        TRACE0("EX22A.DLL Initializing!\n");
        //在这里进行初始化
    }
    else if(dwReason=DLL_PROCESS_DETACH)
    {
        TRACE0("EX22A.DLL Terminating!\n");
            //在这里进行清除工作
    }
     return 1;//成功
}
```

如果程序员没有为DLL模块编写一个DLLMain函数，系统会从其它运行库中引入一个不做任何操作的缺省DLLMain函数版本。在单个线程启动和终止时，DLLMain函数也被调用。正如由dwReason参数所表明的那样。

五、模块句柄

进程中的每个DLL模块被全局唯一的32字节的HINSTANCE句柄标识。进程自己还有一个HINSTANCE句柄。所有这些模块句柄都只有在特定的 进程内部有效，它们代表了DLL或EXE模块在进程虚拟空间中的起始地址。在Win32中，HINSTANCE和HMODULE的值是相同的，这个两种类 型可以替换使用。进程模块句柄几乎总是等于0x400000，而DLL模块的加载地址的缺省句柄是0x10000000。如果程序同时使用了几个DLL模 块，每一个都会有不同的

HINSTANCE值。这是因为在创建DLL文件时指定了不同的基地址，或者是因为加载程序对DLL代码进行了重定位。
模块句柄对于加载资源特别重要。Win32 的FindResource函数中带有一个HINSTANCE参数。EXE和DLL都有其自己的资源。如果应用程序需要来自于DLL的资源，就将此参数指定为DLL的模块句柄。如果需要EXE文件中包含的资源，就指定EXE的模块句柄。

但是在使用这些句柄之前存在一个问题，你怎样得到它们呢？如果需要得到EXE模块句柄，调用带有Null参数的Win32函数GetModuleHandle；如果需要DLL模块句柄，就调用以DLL文件名为参数的Win32函数GetModuleHandle。

六、应用程序怎样找到DLL文件

如果应用程序使用LoadLibrary显式链接，那么在这个函数的参数中可以指定DLL文件的完整路径。如果不指定路径，或是进行隐式链接，Windows将遵循下面的搜索顺序来定位DLL：

1．包含EXE文件的目录，
2．进程的当前工作目录，
3．Windows系统目录，
4．Windows目录，
5．列在Path环境变量中的一系列目录。

这里有一个很容易发生错误的陷阱。如果你使用VC＋＋进行项目开发，并且为DLL模块专门创建了一个项目，然后将生成的DLL文件拷贝到系统目录下，从 应用程序中调用DLL模块。到目前为止，一切正常。接下来对DLL模块做了一些修改后重新生成了新的DLL文件，但你忘记将新的DLL文件拷贝到系统目录 下。下一次当你运行应用程序时，它仍加载了老版本的DLL文件，这可要当心！

七、调试DLL程序

Microsoft 的VC＋＋是开发和测试DLL 的有效工具，只需从DLL项目中运行调试程序即可。当你第一次这样操作时，调试程序会向你询问EXE文件的路径。此后每次在调试程序中运行DLL时，调试 程序会自动加载该EXE文件。然后该EXE文件用上面的搜索序列发现DLL文件，这意味着你必须设置Path环境

变量让其包含DLL文件的磁盘路径，或者 也可以将DLL文件拷贝到搜索序列中的目录路径下。

八、DLL分配的内存需要用dll提供的API释放

附录(B)
I、memwatch的使用说明

1 介绍

MemWatch由 Johan Lindh 编写，是一个开放源代码 C 语言内存错误检测工具。MemWatch支持 ANSI C，它提供结果日志纪录，能检测双重释放（double-free）、错误释放（erroneous free）、内存泄漏（unfreed memory）、溢出(Overflow)、下溢(Underflow)等等。

1.1 MemWatch的内存处理

MemWatch将所有分配的内存用0xFE填充，所以，如果你看到错误的数据是用0xFE填充的，那就是你没有初始化数据。例外是calloc()，它会直接把分配的内存用0填充。

MemWatch将所有已释放的内存用0xFD填充(zapped with 0xFD).如果你发现你使用的数据是用0xFD填充的，那你就使用的是已释放的内存。在这种情况，注意MemWatch会立即把一个"释放了的块信息"填在释放了的数据前。这个块包括关于内存在哪儿释放的信息，以可读的文本形式存放，格式为"FBI<counter>filename(line)"。如:"FBI<267>test.c(12)".使用FBI会降低free()的速度，所以默认是关闭的。使用mwFreeBufferInfo(1)开启。

为了帮助跟踪野指针的写情况，MemWatch能提供no-mans-land（NML）内存填充。no-mans-land将使用0xFC填充.当no-mans-land开启时，MemWatch转变释放的内存为NML填充状态。

1.2初始化和结束处理

一般来说，在程序中使用MemWatch的功能，需要手动添加mwInit()进行初始化，并用对应的mwTerm ()进行结束处理。

当然，如果没有手动调用mwInit()，MemWatch能自动初始化.如果是这种情形，memwatch会使用atext()注册mwTerm()用于atexit-queue. 对于使用自动初始化技术有一个告诫;如果你手动调用atexit()以进行清理工作，memwatch可能在你的程序结束前就终止。为了安全起见，请显式使用mwInit()和mwTerm().

涉及的函数主要有：

mwInit()　mwTerm()　mwAbort()

1.3 MemWatch的I/O 操作

对于一般的操作，MemWatch创建memwatch.log文件。有时，该文件不能被创建;MemWatch会试图创建memwatNN.log文件，NN在01~99之间。

如果你不能使用日志，或者不想使用，也没有问题。只要使用类型为"void func(int c)"的参数调用mwSetOutFunc()，然后所有的输出都会按字节定向到该函数.

当ASSERT或者VERIFY失败时，MemWatch也有Abort/Retry/Ignore处理机制。默认的处理机制没有I/O操作，但是会自动中断程序。你可以使用任何其他Abort/Retry/Ignore的处理机制,只要以参数"void func(int c)”调用mwSetAriFunc()。

涉及的函数主要有：

mwTrace()　mwPuts()　mwSetOutFunc()　mwSetAriFunc() mwSetAriAction()　mwAriHandler()　mwBreakOut()

1.4 MemWatch对C++的支持

可以将MemWatch用于C++,但是不推荐这么做。请详细阅读memwatch.h中关于对C++的支持。

2 使用

2.1 为自己的程序提供MemWatch功能

- 在要使用MemWatch的.c文件中包含头文件"memwatch.h"
- 使用GCC编译（注意：不是链接）自己的程序时，加入-DMEMWATCH -DMW_STDIO
- Windows平台vs编译器可以在预处理器宏定义加入MEMWATCH和DMW_STDIO两项

如：gcc -DMEMWATCH -DMW_STDIO –o test.o –c test1.c

2.2 使用MemWatch提供的功能

1）在程序中常用的MemWatch功能有：

```
mwTRACE ( const char* format_string, ... );
//或
TRACE ( const char* format_string, ... );

mwASSERT ( int, const char*, const char*, int );
//或
ASSERT ( int, const char*, const char*, int );

mwVERIFY ( int, const char*, const char*, int );
//或
VERIFY ( int, const char*, const char*, int );

mwPuts ( const char* text );

/*ARI机制*/
    mwSetAriFunc(int (*func)(const char *));
    mwSetAriAction(int action);
    mwAriHandler ( const char* cause )

mwSetOutFunc (void (*func)(int));
mwIsReadAddr(const void *p, unsigned len );
mwIsSafeAddr(void *p, unsigned len );
mwStatistics ( int level );
mwBreakOut ( const char* cause);
```

2）mwTRACE，mwASSERT，mwVERIFY和mwPuts顾名思义，就不再赘述。仅需要注意的是，Memwatch定义了宏TRACE， ASSERT 和 VERIFY. 如果你已使用同名的宏,memwatch2.61及更高版本的memwatch不会覆盖你的定义。MemWatch2.61及以后，定义了mwTRACE, mwASSERT 和 mwVERIFY宏，这样，你就能确定使用的是memwatch的宏定义。2.61版本前的memwatch会覆盖已存在的同名的TRACE, ASSERT 和 VERIFY定义。

当然，如果你不想使用MemWatch的这几个宏定义，可以定义 MW_NOTRACE, MW_NOASSERT 和 MW_NOVERIFY宏，这样MemWatch的宏定义就不起作用了。所有版本的memwatch都遵照这个规则。

3）ARI机制即程序设置的“Abort, Retry, Ignore选择陷阱。

mwSetAriFunc：

设置“Abort, Retry, Ignore”发生时的MemWatch调用的函数.当这样设置调用的函数地址时，实际的错误消息不会打印出来，但会作为一个参数进行传递。

如果参数传递NULL，ARI处理函数会被再次关闭。当ARI处理函数关闭后， meewatch会自动调用有mwSetAriAction()指定的操作。
正常情况下，失败的ASSERT() or VERIFY()会中断你的程序。但这可以通过mwSetAriFunc()改变，即通过将函数"int myAriFunc(const char *)"传给它实现。你的程序必须询问用户是否中断，重试或者忽略这个陷阱。返回2用于Abort， 1用于Retry，或者0对于Ignore。注意retry时，会导致表达式重新求值.

MemWatch有个默认的ARI处理器。默认是关闭的，但你能通过调用mwDefaultAri()开启。注意这仍然会中止你的程序除非你定义MEMWATCH_STDIO允许MemWatch使用标准C的I/O流。
同时，设置ARI函数也会导致MemWatch不将ARI的错误信息写向标准错误输出，错误字符串而是作为'const char *’ 参数传递到ARI函数.

mwSetAriAction：

如果没有ARI处理器被指定，设置默认的ARI返回值。默认是MW_ARI_ABORT

mwAriHandler：

这是个标准的ARI处理器，如果你喜欢就尽管用。它将错误输出到标准错误输出，并从标准输入获得输入。

mwSetOutFunc：

将输出转向调用者给出的函数(参数即函数地址)。参数为NULL，表示把输出写入日志文件memwatch.log.

mwIsReadAddr:

检查内存是否有读取的权限

mwIsSafeAddr:

检查内存是否有读、写的权限

mwStatistics：

设置状态搜集器的行为。对应的参数采用宏定义。

```
#define MW_STAT_GLOBAL  0       /* 仅搜集全局状态信息 */
#define MW_STAT_MODULE  1       /* 搜集模块级的状态信息 */
#define MW_STAT_LINE    2       /* 搜集代码行级的状态信息 */
#define MW_STAT_DEFAULT 0       /* 默认状态设置 */
```

mwBreakOut:

当某些情况MemWatch觉得中断(break into)编译器更好时，就调用这个函数.如果你喜欢使用MemWatch,那么可以在这个函数上设置执行断点。其他功能的使用，请参考源代码的说明。

2.3分析日志文件

日志文件memwatch.log中包含的信息主要有以下几点：

- 测试日期
- 状态搜集器的信息
- 使用MemWatch的输出函数或宏（如TRACE等）的信息。
- MemWatch捕获的错误信息
- 内存使用的全局信息统计，包括四点：

1）分配了多少次内存
2）最大内存使用量

3）分配的内存总量
4）为释放的内存总数

MemWatch捕获的错误记录在日志文件中的输出格式如下：
message: <sequence-number> filename(linenumber), information

2.4 注意事项

mwInit()和mwTerm()是对应的.所以使用了多少次mwInit()，就需要调用多少次mwTerm()用于终止MemWatch.

如果在流程中捕获了程序的异常中断，那么需要调用mwAbort()而不是mwTerm()。即使有显示的调用mwTerm()，mwAbort()也将终止MemWatch。

MemWatch不能确保是线程安全的。如果你碰巧使用Wind32或者你使用了线程，作为2.66，是初步支持线程的。定义WIN32或者MW_PTHREADS以明确支持线程。这会导致一个全局互斥变量产生，同时当访问全局内存链时，MemWatch会锁定互斥变量，但这远不能证明是线程安全的。

3 结论

从MemWatch的使用可以得知，无法用于内核模块。因为MemWatch自身就使用了应用层的接口，而不是内核接口。但是，对于普通的应用层程序，我认为还是比较有用，并且是开源的，可以自己修改代码实现；它能方便地查找内存泄漏，特别是提供的接口函数简单易懂，学习掌握很容易，对应用层程序的单元测试会较适用。

II、Linux C 编程内存泄露检测工具(一)：mtrace

前言

所有使用动态内存分配(dynamic memory allocation)的程序都有机会遇上内存泄露(memory leakage)问题，在Linux里有三种常用工具来检测内存泄露的情况，包括：

1. mtrace
2. dmalloc
3. memwatch

1. mtrace

mtrace是三款工具之中是最简单易用的，mtrace是一个C函數，在<mcheck.h>里声明及定义，函数原型为：

```
void mtrace(void);
```

其实mtrace是类似malloc_hook的 malloc handler，只不过mtrace的handler function已由系统为你写好，但既然如此，系统又怎么知道你想将malloc/free的记录写在哪里呢？为此，调用mtrace()前要先设置MALLOC_TRACE环境变量：

```
#include <stdlib.h>
//....

setenv("MALLOC_TRACE", "output_file_name", 1);

//...
```

「output_file_name」就是储存检测结果的文件的名称。
但是检测结果的格式是一般人无法理解的，而只要有安装mtrace的话，就会有一名为mtrace的Perl script，在shell输入以下指令：

mtrace [binary] output_file_name

就会将output_file_name的内容转化成能被理解的语句，例如「No memory leaks」，「0x12345678 Free 10 was never alloc」诸如此类。
例如以下有一函数：(暂且放下single entry single exit的原则)

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <errno.h>
#include <mcheck.h>

int main()
{
    char *hello;

    setenv("MALLOC_TRACE", "output", 1);
    mtrace();

    if ((hello = (char *) malloc(sizeof(char))) == NULL) {
        perror("Cannot allocate memory.");
        return -1;
    }

   return 0;
}
```

执行后，再用mtrace 将结果输出：

```
- 0x08049670 Free 3 was never alloc'd 0x42029acc
- 0x080496f0 Free 4 was never alloc'd 0x420dc9e9
- 0x08049708 Free 5 was never alloc'd 0x420dc9f1
- 0x08049628 Free 6 was never alloc'd 0x42113a22
- 0x08049640 Free 7 was never alloc'd 0x42113a52
- 0x08049658 Free 8 was never alloc'd 0x42113a96

Memory not freed:
-----------------
   Address     Size     Caller
0x08049a90      0x1  at 0x80483fe
```

最后一行标明有一个大小为1 byte的内存尚未释放，大概是指「hello」吧。

若我们把该段内存释放：

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <mcheck.h>


int main()
{

    char *hello;

    setenv("MALLOC_TRACE", "output", 1);
    mtrace();
```

```
    if ((hello = (char *) malloc(sizeof(char))) == NULL) {
        perror("Cannot allocate memory.");
        return -1;
    }

    free(hello);

    return 0;
}
```

结果如下：

```
- 0x080496b0 Free 4 was never alloc'd 0x42029acc
- 0x08049730 Free 5 was never alloc'd 0x420dc9e9
- 0x08049748 Free 6 was never alloc'd 0x420dc9f1
- 0x08049668 Free 7 was never alloc'd 0x42113a22
- 0x08049680 Free 8 was never alloc'd 0x42113a52
- 0x08049698 Free 9 was never alloc'd 0x42113a96
No memory leaks.
```

mtrace的原理是记录每一对malloc-free的执行，若每一个malloc都有相应的free，则代表没有内存泄露，对于任何非malloc/free情况下所发生的内存泄露问题，mtrace并不能找出来。

III、Linux C 编程内存泄露检测工具(二)：memwatch

Memwatch简介

在三种检测工具当中，设置最简单的算是memwatch，和dmalloc一样，它能检测未释放的内存、同一段内存被释放多次、位址存取错误及不当使用未分配之内存区域。请往http://www.linkdata.se/sourcecode.html 下载最新版本的Memwatch。

安装及使用memwatch

很幸运地，memwatch根本是不需要安装的，因为它只是一组C程序代码，只要在你程序中加入memwatch.h，编译时加上-DMEMWATCH -DMW_STDIO及memwatch.c就能使用memwatch，例如：

```
gcc  -DMEMWATCH  -DMW_STDIO  test.c  memwatch.c  -o  test
```

memwatch输出结果

memwatch 的输出文件名称为memwatch.log，而且在程序执行期间，所有错误提示都会显示在stdout上，如果memwatch未能写入以上文件，它会尝 试写入memwatchNN.log，而NN介于01至99之间，若它仍未能写入memwatchNN.log，则会放弃写入文件。

我们引用第一篇(mtrace)中所使用过的有问题的代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <memwatch.h>


int main()
{
     char *hello;

    setenv("MALLOC_TRACE", "output", 1);
    mtrace();
    if ((hello = (char *) malloc(sizeof(char))) == NULL) {
        perror("Cannot allocate memory.");
        return -1;
    }

    return 0;
}
```

然后在shell中输入以下编译指令：

```
gcc  -DMEMWATCH  -DMW_STDIO  test.c  memwatch.c  -o  test
```

memwatch.log的内容如下：

```
============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =============

Started at Sat Jun 26 22:48:47 2004

Modes: __STDC__ 32-bit mwDWORD==(unsigned long)
mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32


Stopped at Sat Jun 26 22:48:47 2004

    unfreed: <1> test.c(9), 1 bytes at 0x805108c    {FE .. .. .. .. .. .. .. .. .. .. .. .. .. .. .}

Memory usage statistics (global):
 N)umber of allocations made: 1
 L)argest memory usage      : 1
```

```
T)otal of all alloc() calls: 1
U)nfreed bytes totals      : 1
```

文件指出，在test.c被执行到第9行时所分配的内存仍未被释放，该段内存的大小为1 byte。

Memwatch使用注意

Memwatch 的优点是无需特别配置，不需安装便能使用，但缺点是它会拖慢程序的运行速度，尤其是释放内存时它会作大量检查。但它比mtrace和dmalloc多了 一项功能，就是能模拟系统内存不足的情况，使用者只需用mwLimit(long num_of_byte)函数来限制程式的heap memory大小(以byte单位)。

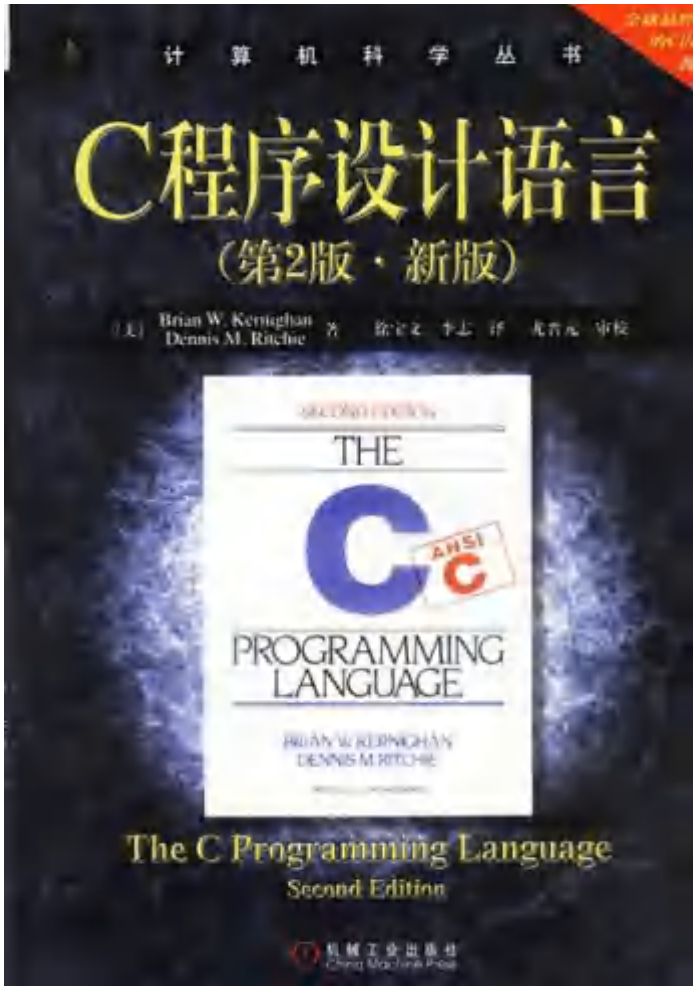
最详细的使用说明(包括优点缺点，运行原理等)已在README中列出，本人强烈建议各位读者参考该文件。

附录（B）

拓展阅读

作者: K. N. King
出版社: 人民邮电出版社
出版年: 2007-11
页数: 408
定价: 55.00元
丛书: 图灵计算机科学丛书
ISBN: 9787115167071

作者: （美）Brian W. Kernighan / （美）Dennis M. Ritchie
出版社: 机械工业出版社
副标题: 第 2 版·新版
原作名: The C Programming Language
译者: 徐宝文 / 李志译 / 尤晋元审校
出版年: 2004-1-1
页数: 258
定价: 30.00元



装帧: 平装
丛书: 计算机科学丛书
ISBN: 9787111128069