Easy 搞定

C 语言

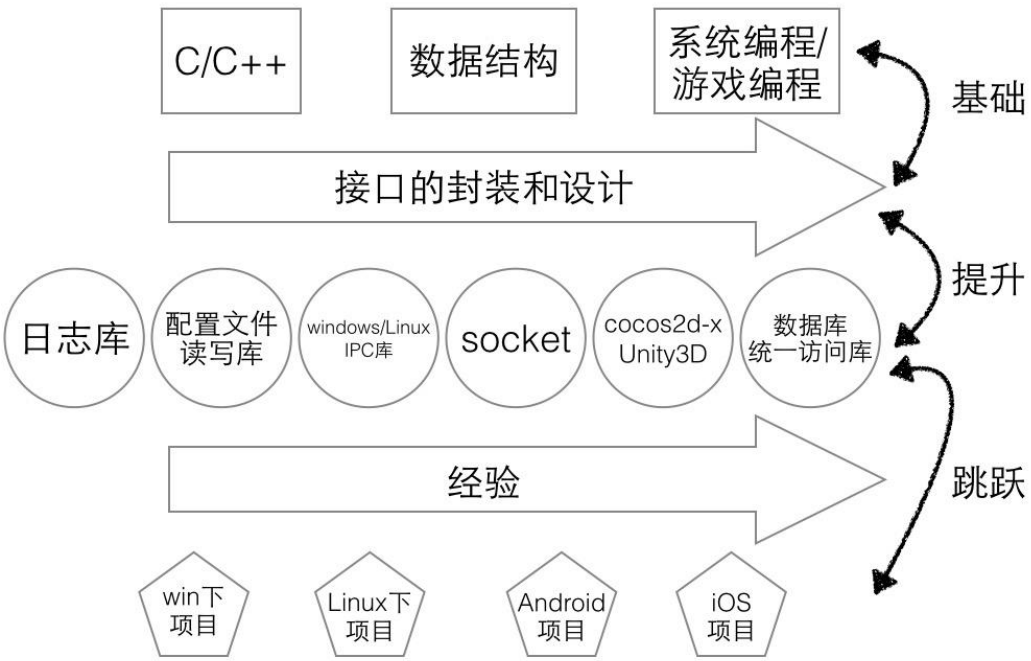
(提高篇)

0 精神与契约

企业需要能干活的人，需要能上战场的兵。

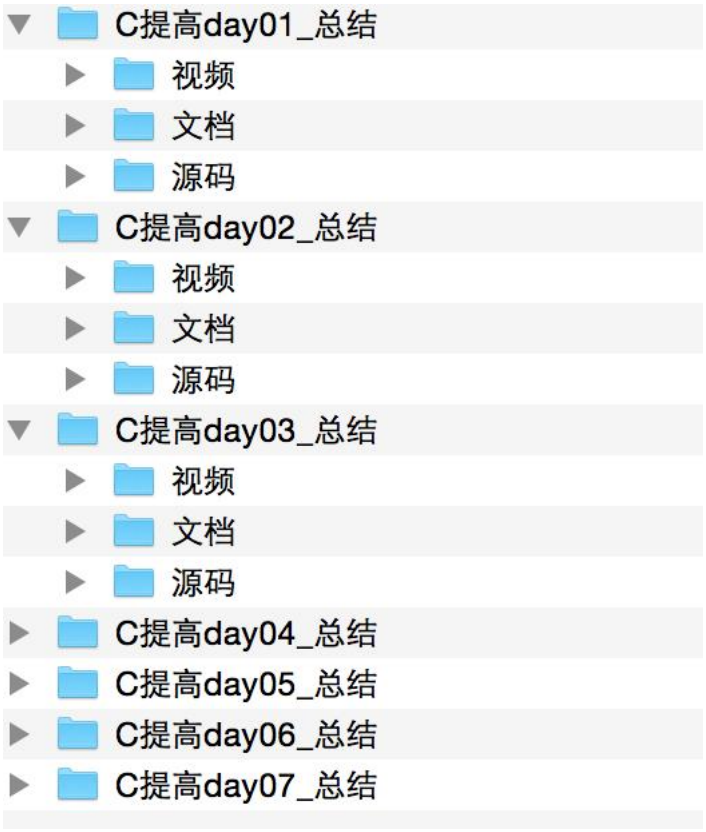
0.1 如何成为一个对企业有价值的人？

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。

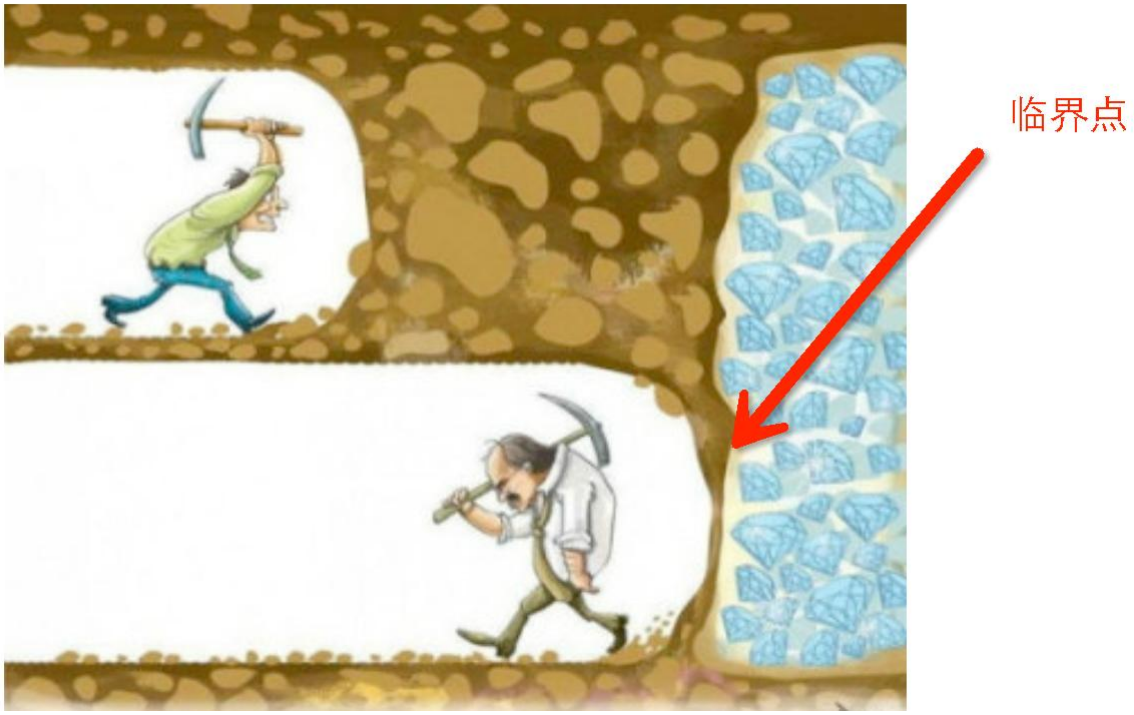


0.2 战前准备：

- 资料管理


C提高day01_总结
视频
文档
源码
C提高day02_总结
视频
文档
源码
C提高day03_总结
视频
文档
源码
C提高day04_总结
C提高day05_总结
C提高day06_总结
C提高day07_总结

- 工作经验，记录和积累
- 临界点



塑造自己的过程很疼，但你可以得到一个更好的自己。

- 当堂运行
- 动手
- 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，请课前做好预习。

  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。

  如果时间允许，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。

0.3 上战场的能力

0.3.1 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

0.3.2 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

0.3.3 学习标准

```
//第一套api接口
int socketclient_init(void **handle);
int socketclient_send(void *handle, unsigned char *buf, int len);
int socketclient_recv(void *handle, unsigned char *buf, int *len);
int socketclient_destory(void *handle);

//第二套api接口
int socketclient_init2(void **handle);
int socketclient_send2(void *handle, unsigned char *buf, int len);
int socketclient_recv2(void *handle, unsigned char **buf, int *len);
int socketclient_free(unsigned char **buf);
int socketclient_destory2(void **handle);
```

0.3.4 听课标准

```
//选择法排序
// i = 0 时,  j 从 1 到 n 变化
// i = 1 时,  j 从 2 到 n 变化
// ……
for (i = 0; i < 10; i++)
{
    for (j = i + 1; j < 10; j++)
    {
        if (a[i] > a[j]) //升序，大于时才交换
        {
            tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
        }
    }
}
```

1. 内存四区

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

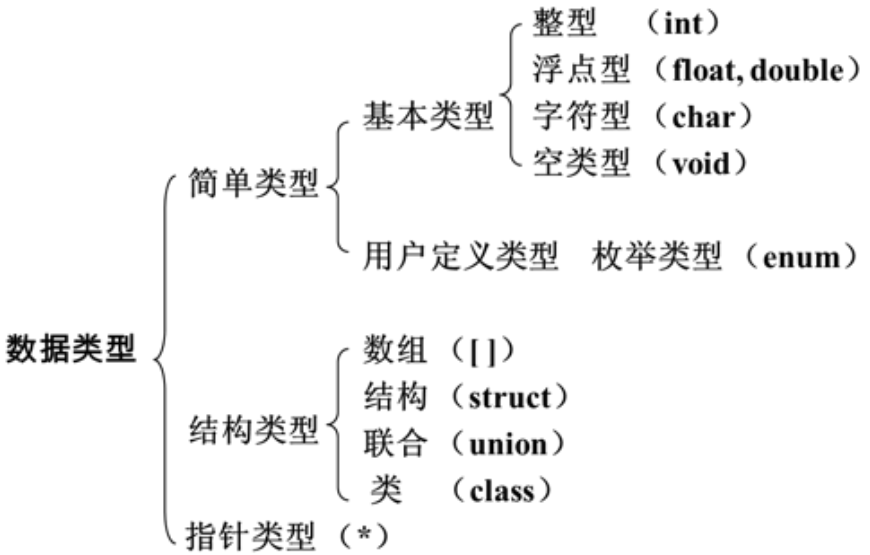
1.1.1 数据类型概念

- “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，才会发现它的本质。

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。
- 注意：数据类型只是模具，编译器并没有分配空间，只有根据类型（模具）创建变量（实物），编译器才会分配空间。

```
#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

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

```c
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

```c
#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

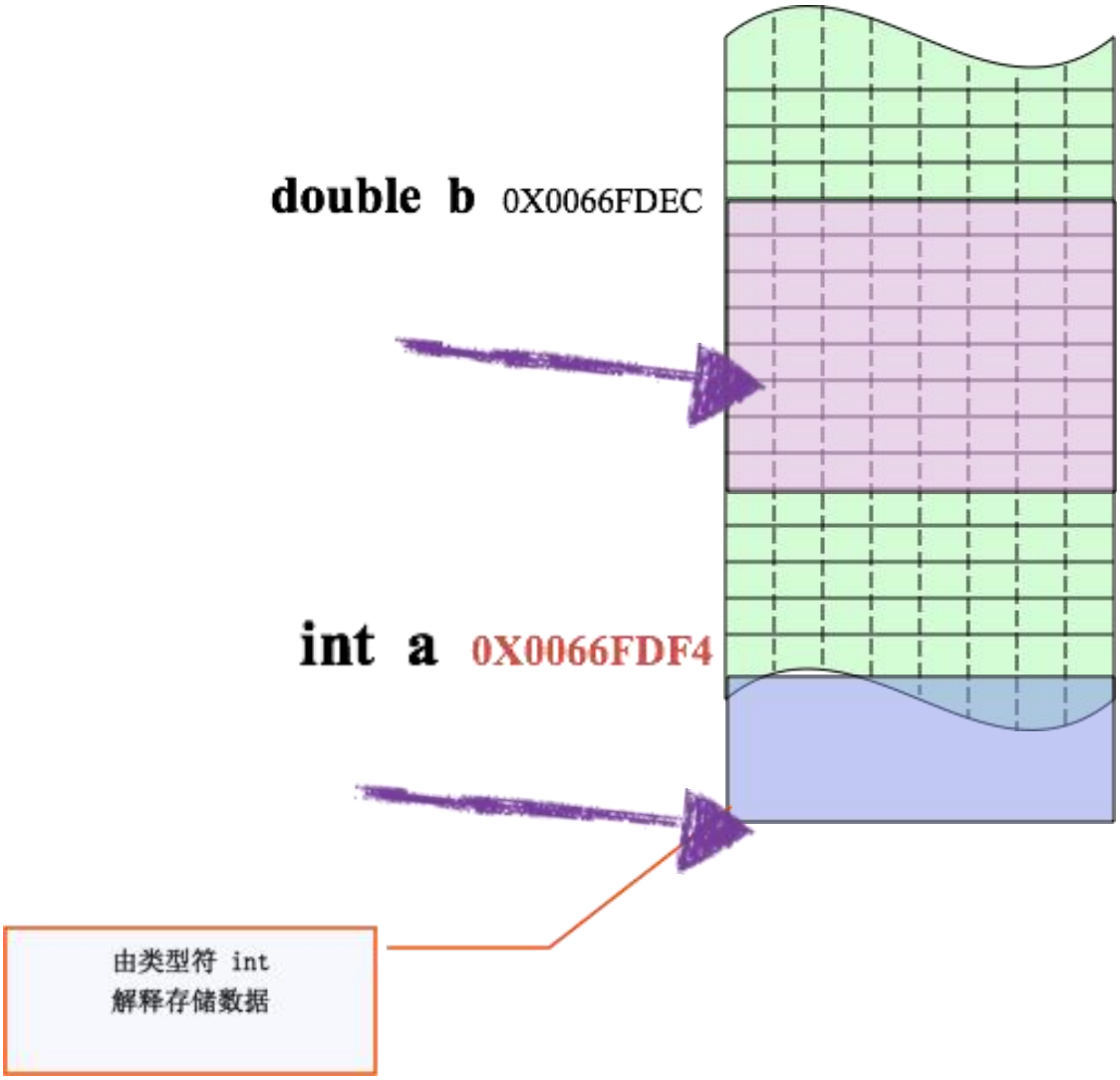
1.2.1 变量的概念

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

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

int main(void)
{
    int i = 0;

    // 通过变量直接操作内存
    i = 10;

    int *p = &i;
    printf("&i:%d\n", &i);
    printf("p:%d\n", p);

    // 通过内存编号间接操作内存
    *p = 100;
    printf("i = %d, *p = %d\n", i, *p);

    system("pause");
    return 0;
}
```

4、数据类型和变量的关系

通过数据类型定义变量

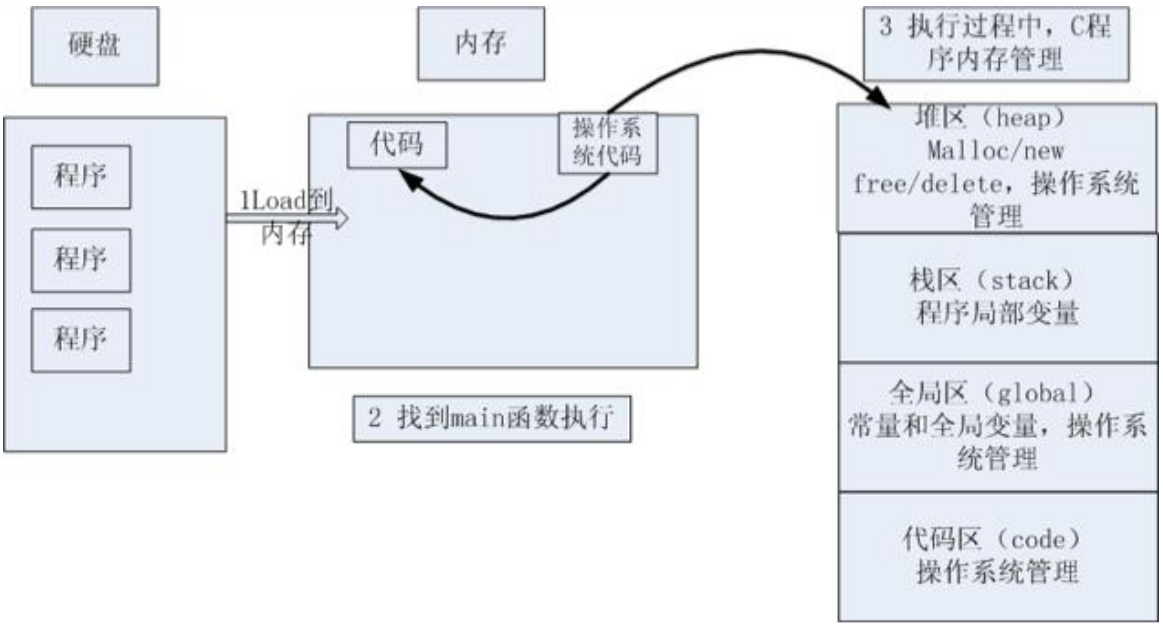
5、总结

1 对内存，可读可写；

2 通过变量往内存读写数据；

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

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。

3、操作系统找到 main 函数入口执行

建立正确程序运行内存布局图是学好 C 的关键

1.3.1 栈区和堆区

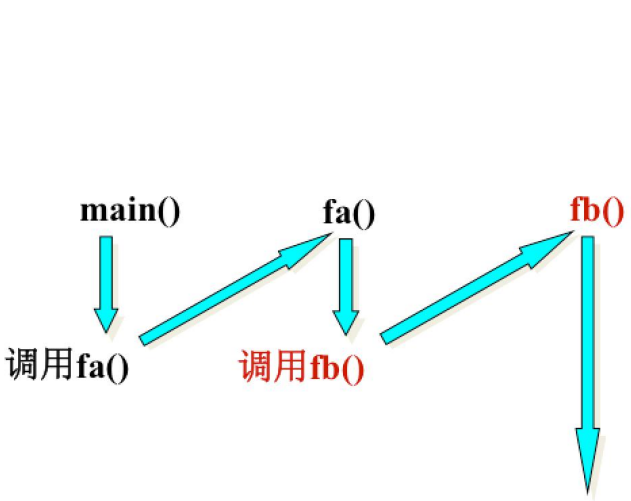
```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//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 ;
```

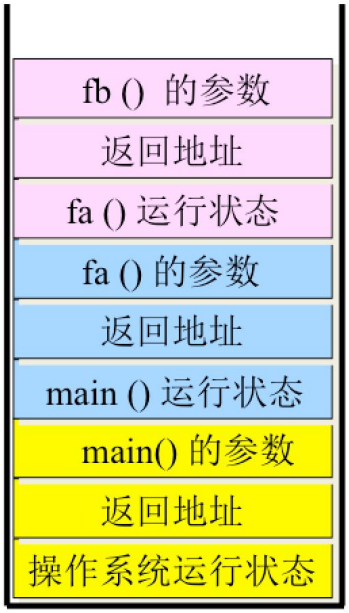
```
    }
    strcpy(tmp, "111222"); //向 tmp 做指向的内存空间中 copy 数据

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

    return 0;
}
```

1.3.2 全局区

```
#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



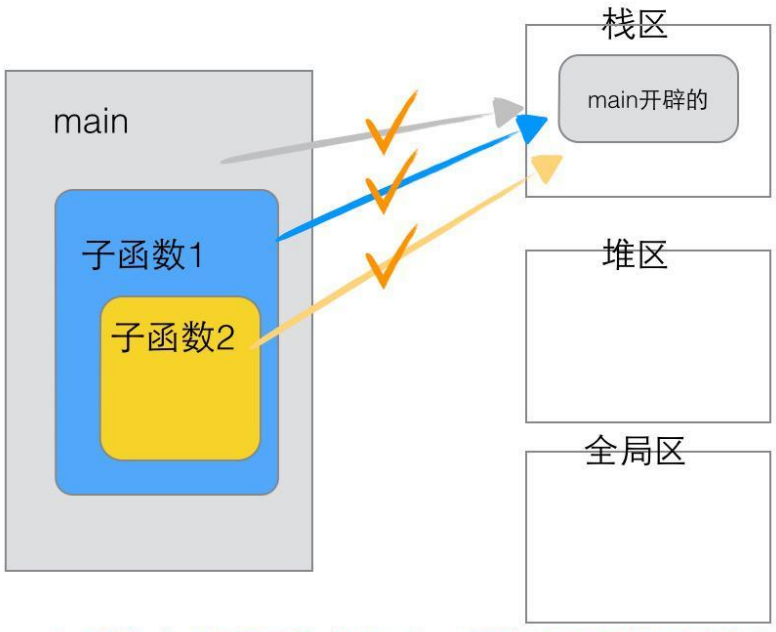


栈

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

int fa(int a)
{
  fb();
  return 0;
}

int main(void)
{
  int a = 3;
  fa(a);
  return 0;
}
```

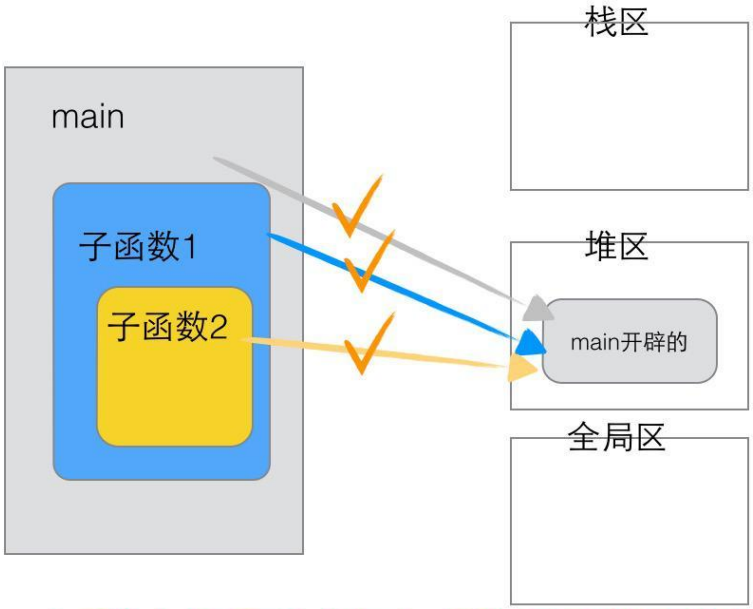
1.5 函数调用变量传递分析

（1）



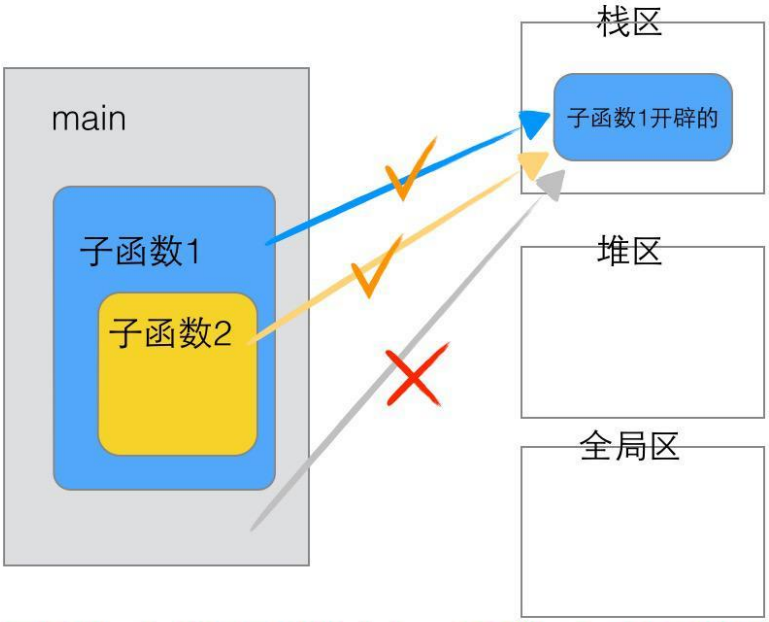
main函数在栈区开辟的内存，所有子函数均可以使用

(2)



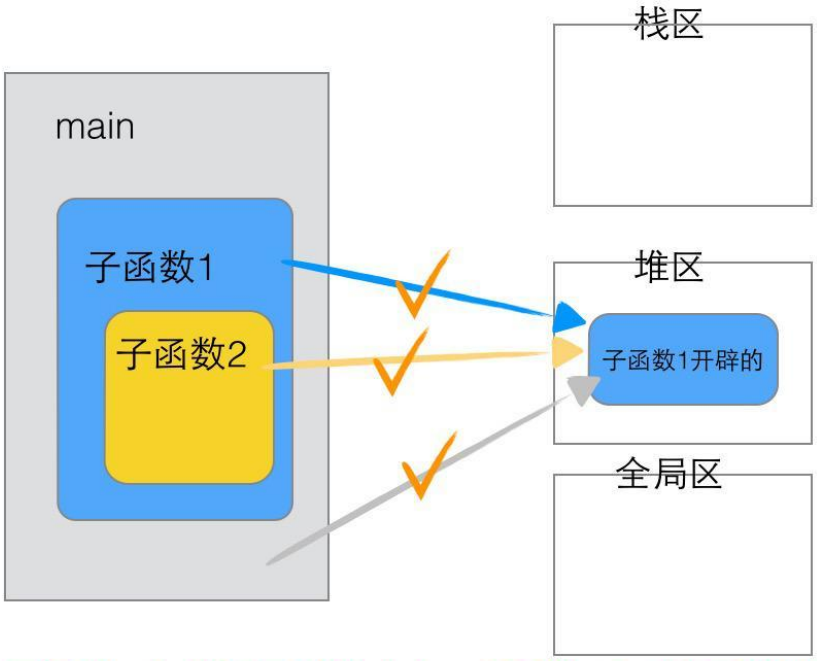
main函数在堆区开辟的内存，所有子函数均可以使用

(3)



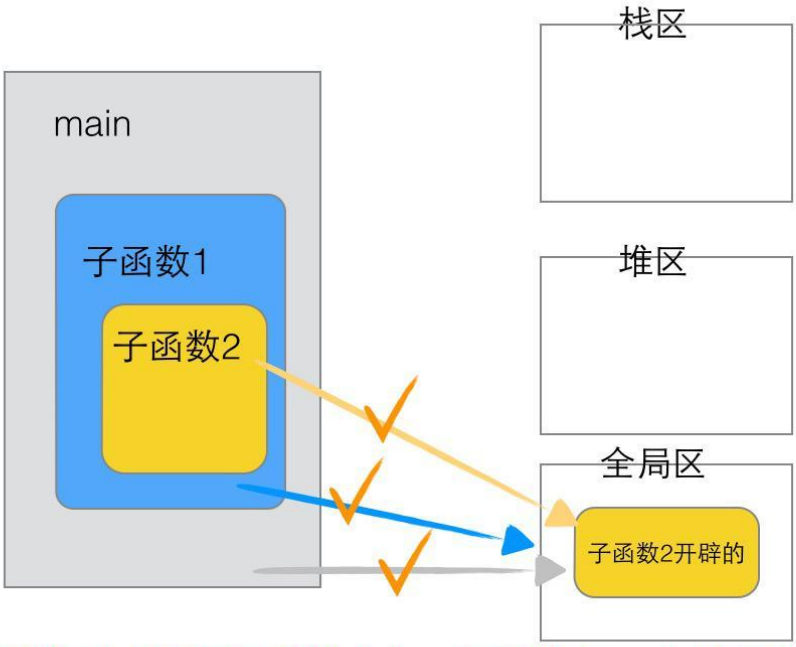
子函数1在栈区开辟的内存，子函数1和2均可以使用

(4)

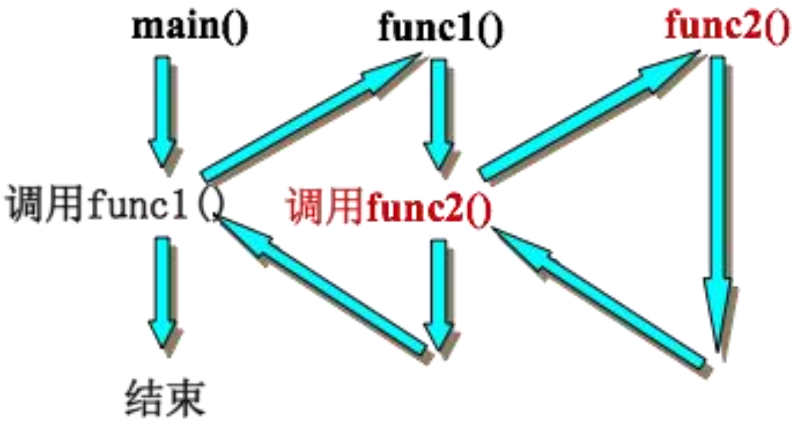


子函数1在栈区开辟的内存，子函数1和2均可以使用

(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



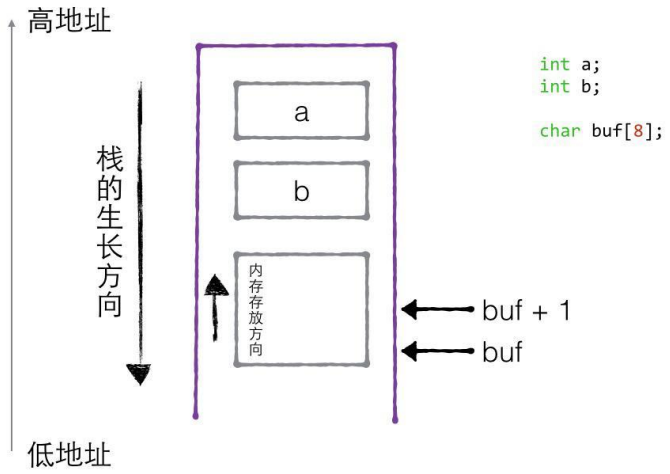
1. main 函数中可以在栈/堆/全局分配内存，都可以被 func1 和 func2 使用

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

1.6 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;
    *p = 20;

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

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

    return 0;
}
```

2. 指针

强化 1：指针是一种数据类型

1） 指针变量也是一种变量，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，写内存）

*p 放在等号的右边取值（从内存获取值，读内存）

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，含义不同切记!

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);

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）。指针指向谁，就把谁的地址赋值给指针。

```
#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 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;
}
```

```c
//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

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



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

```
}

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

```
#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++;
   }
}
```

```c
//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。

```c
#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
```

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. 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' （数字 0 和字符 '\0' 等价）。

3.1.2 数组法和指针法操作字符串

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

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

    //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

```c
#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。

```c
#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);

}

//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>
```

```c
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

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

```
#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：

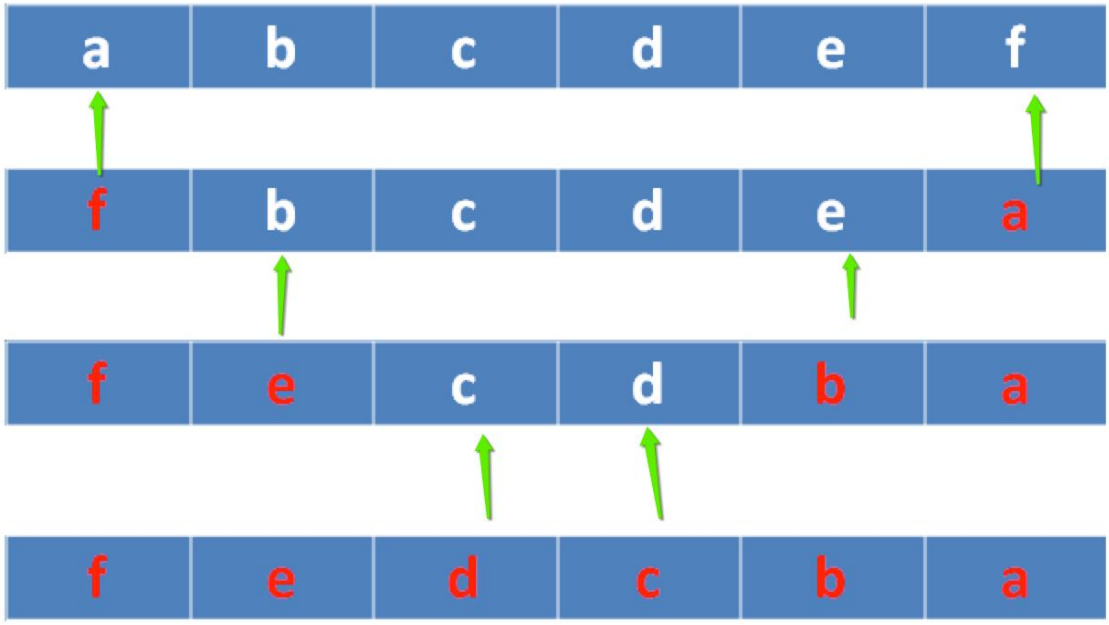
有一个字符串开头或结尾含有 n 个空格 （“ abcdefgdddd ”），欲去掉前后空格，返回一个新字符串。

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
```

```c
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;
}
```

```c
//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：

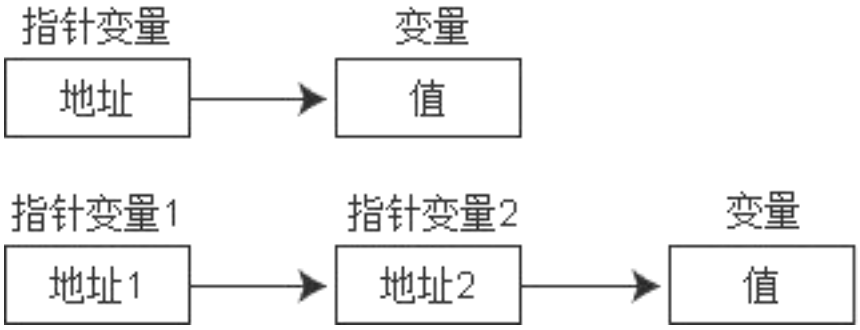
```
"key1 = value1"
"key2 =       value2"
"key3  = value3"
"key4        = value4"
"key5   =   "
"key6   ="

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

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

   return 0;
}
```

4. 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”。



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);
```

三种二级指针内存模型图

练习：通过下列代码画出三种二级指针模型的内存四区图

```
#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 *)); //char *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：

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);
```

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(第二种内存模型)

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

int spitString(const char *str, char c, char buf[10][30], int *count);

第二个 API(第三种内存模型)

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

int spitString2(const char *str, char c, char **myp /*in*/, int *count);

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.

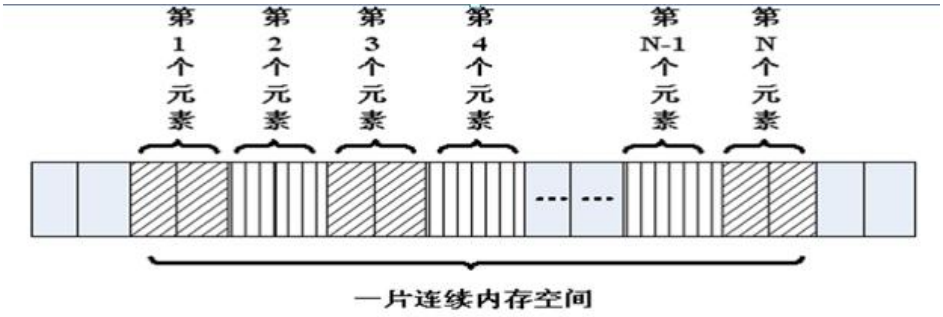
3, 编写正确的测试用例.

5. 多维数组

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

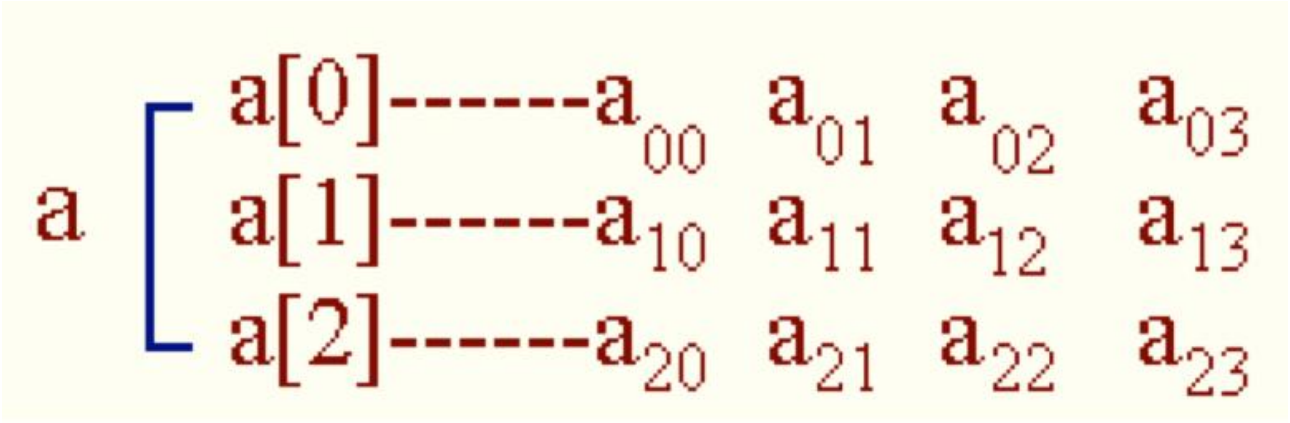
所谓多维数组就是二维和大于二维的数组,在 C 语言中并不直接支持多维数组,包括二维数 组。多维数组的声明是使用一维数组的嵌套声明实现的。一个一维数组的每个元素又被声明为一 维数组,从而构成二维数组,可以说二维数组是特殊的一维数组。

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?

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];

```
#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;
```

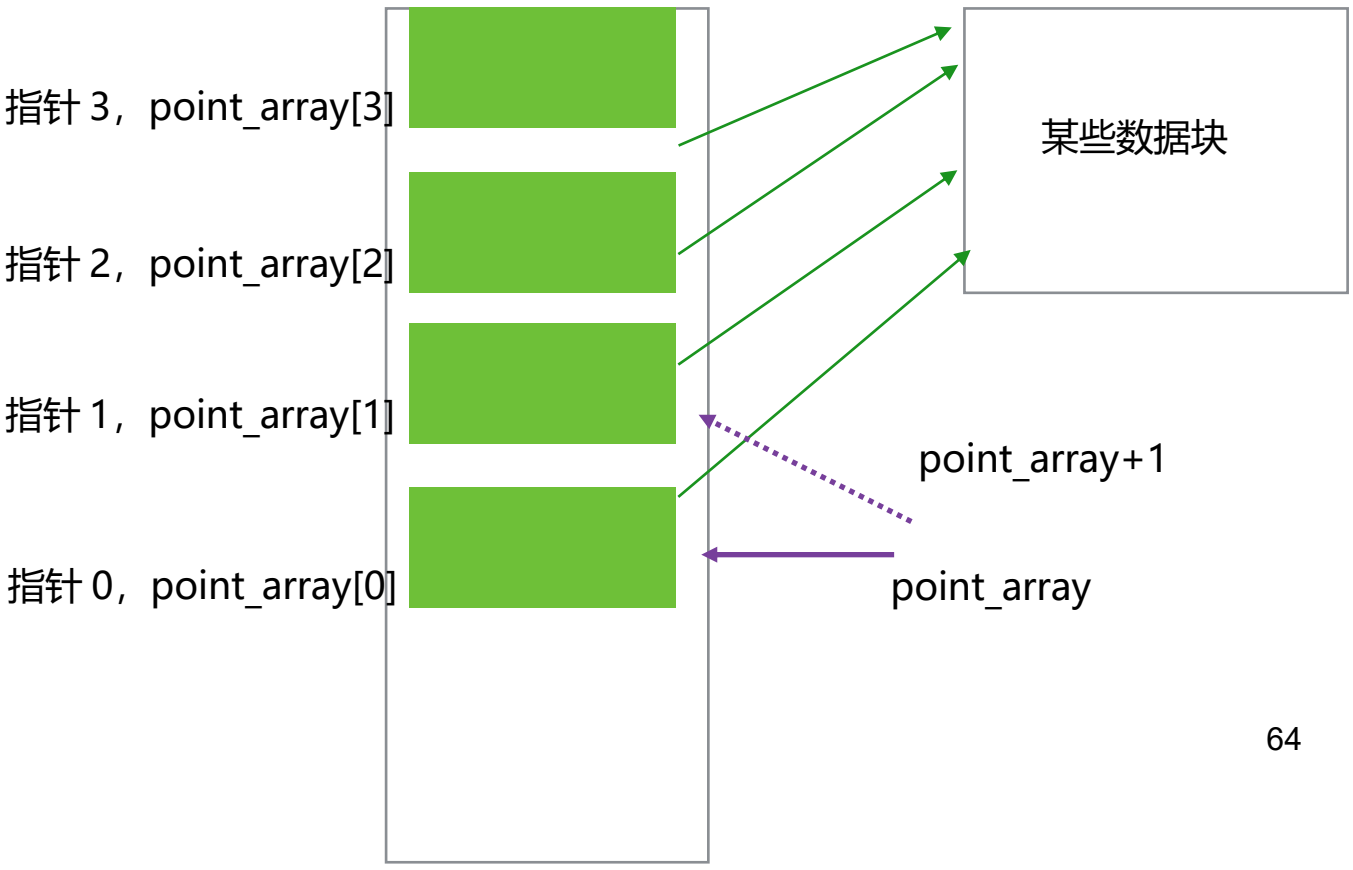
```
    for (i=0; i<5; i++)
    {
        array[i] = i;
    }

    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];
```

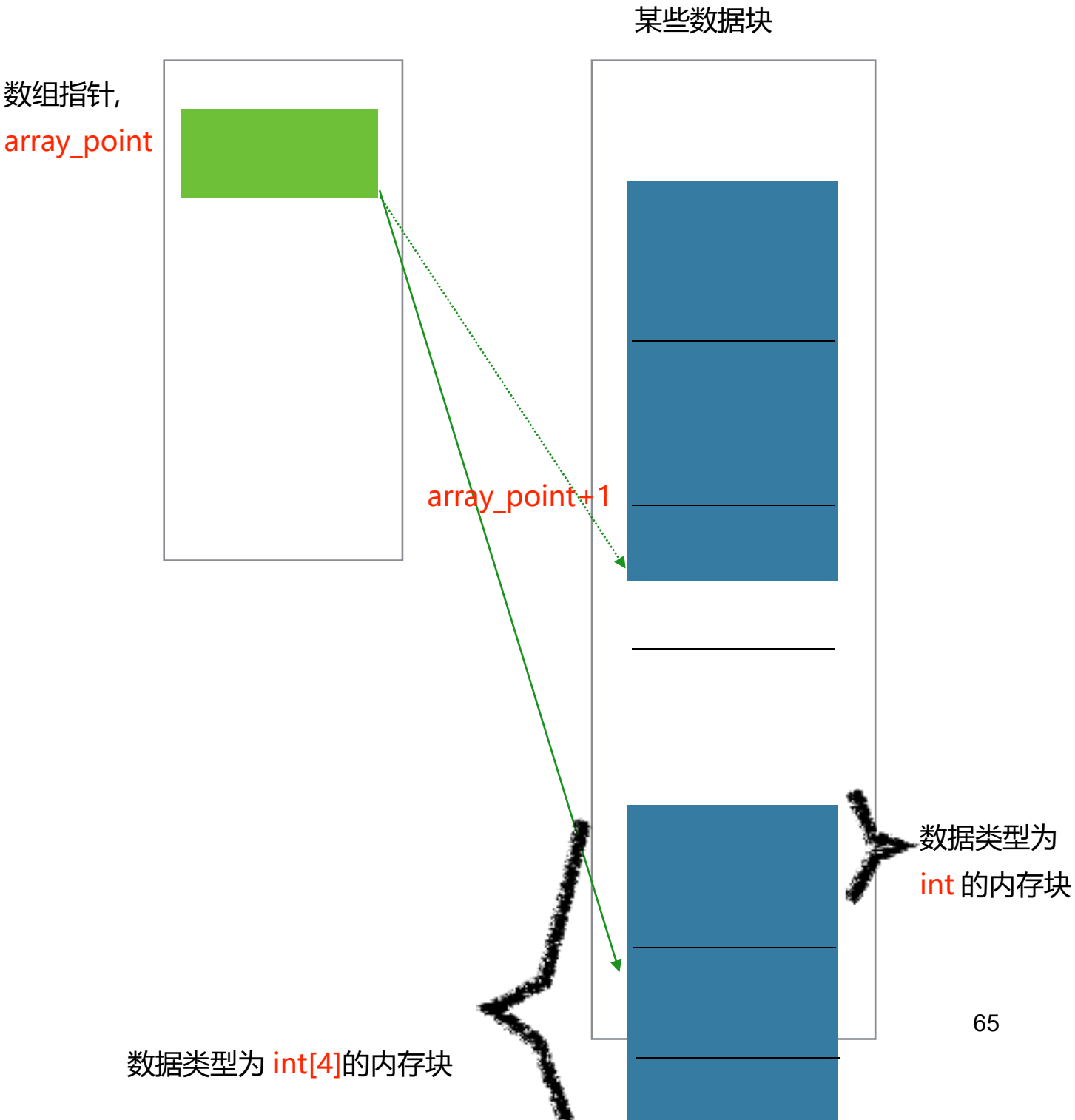
指针数组，是一个数组, 里面的每一个元素都是一个指针。（多个指针）



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。

5.33 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]**

```
#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 = 60

   //多维数组名的本质就是一个指向第一个维度的数组的指针
   //步长为第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
     pArray = a;
   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;
}
```

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

```
#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

```
#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]);
        }
    }
}

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;
}
```

7. 结构体

7.1 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;
}
```

7.2 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];
};

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;
}
```

7.3 结构体数组

```
#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;
}
```

7.4 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;
}
```

7.5 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++)
```

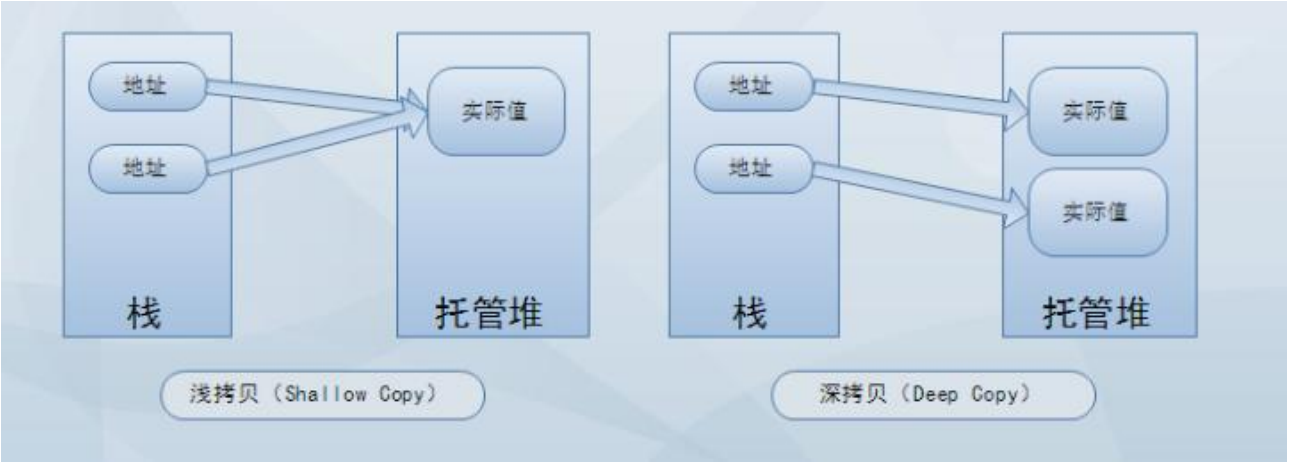
```
        {
            if (myp[j] != NULL)
            {
                free(myp[j]);
            }
        }
        free(myp);
        p[i].student_name = NULL;
    }
  }
  free(p);
}

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

7.6 有关浅拷贝深拷贝问题




```
#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;
}
```

7.7 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 ;
}
```

7.8 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。

7.8.1 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。

7.8.2 内存对齐的原则

默认情况下，数据成员的对齐规则(以最大的类型字节为单位)。

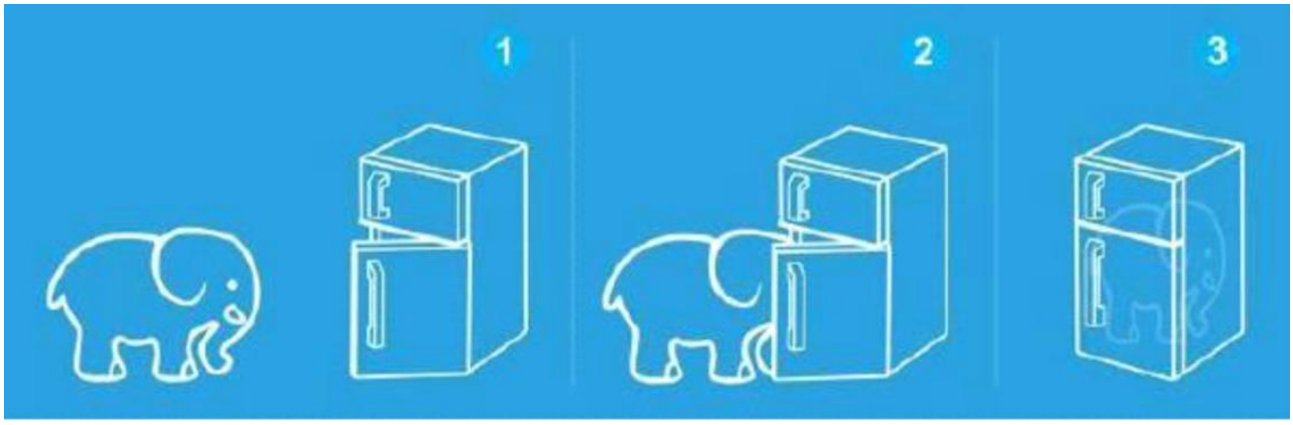
当然，字节对齐可以通过程序控制，采用指令：

```
#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
```

8. 文件操作

8.1 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

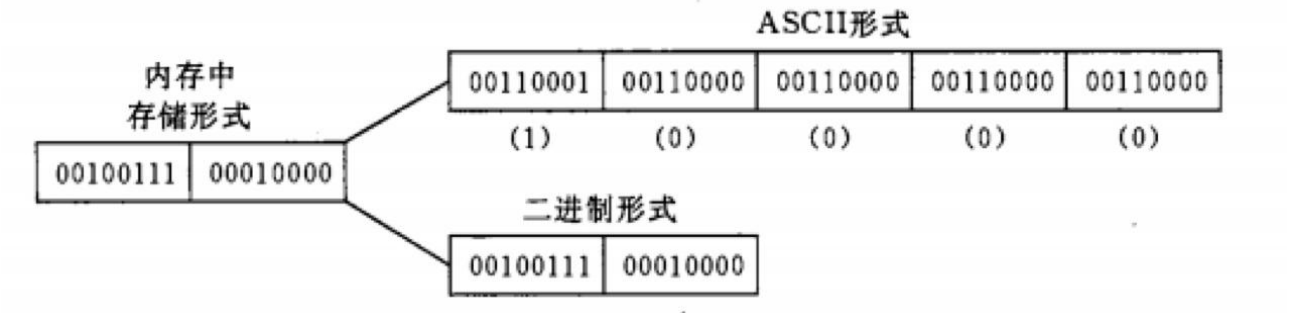
3)打开文件

4)文件读写

5)关闭文件

8.2 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

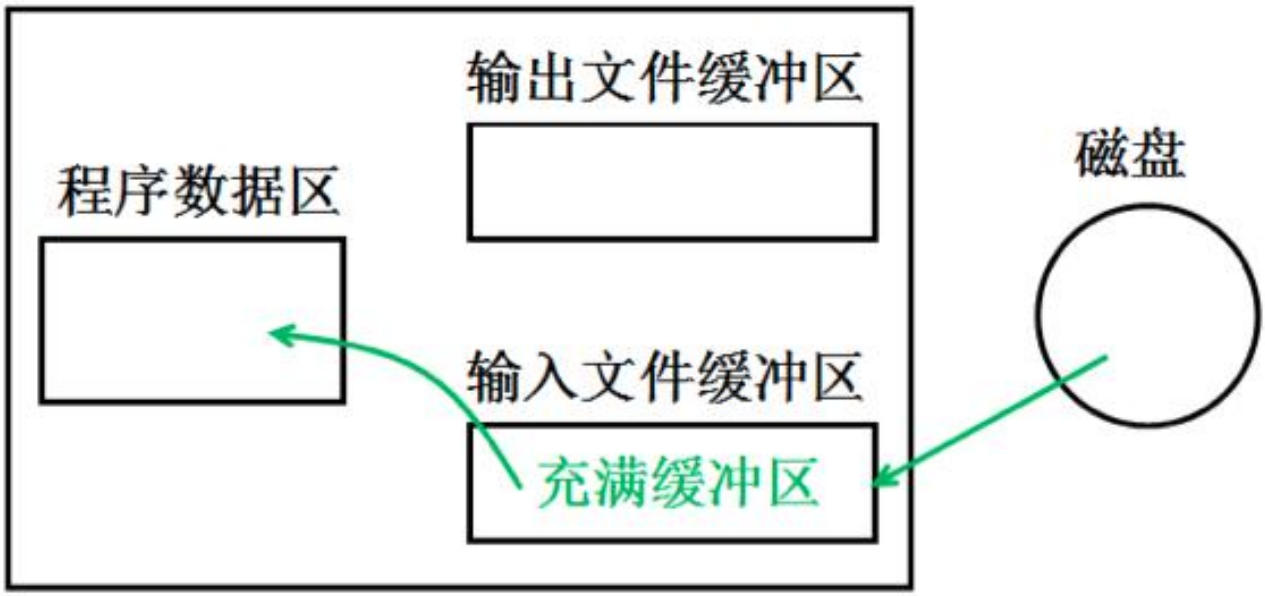
1)文件缓冲区

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)

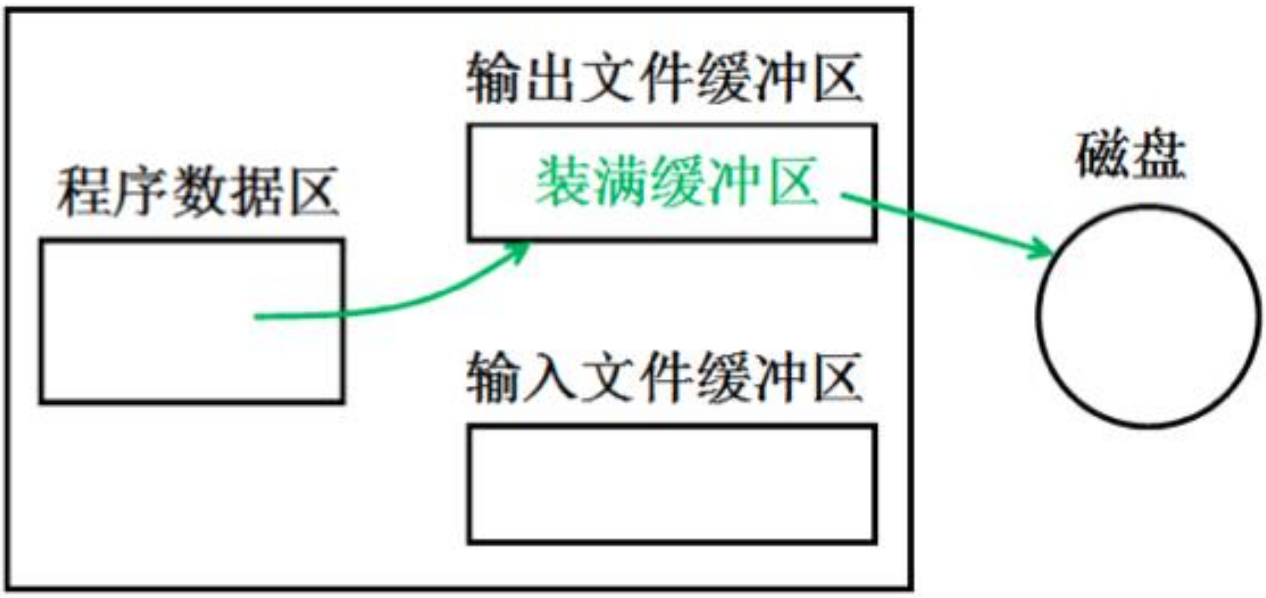
2)输入输出流

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;
```

8.3 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。

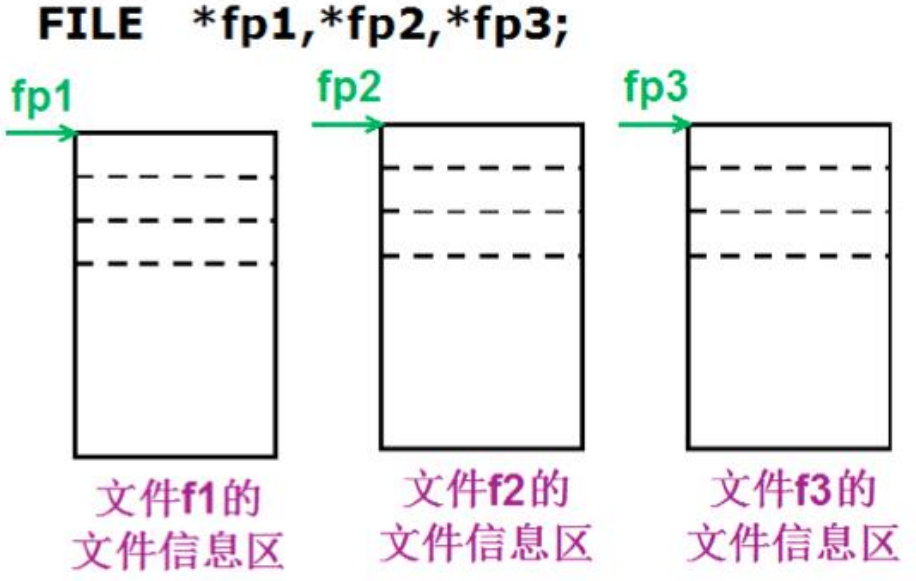
声明 FILE 结构体类型的信息包含在头文件“stdio.h”中
一般设置一个指向 FILE 类型变量的指针变量,然后通过它来引用这些 FILE 类型变量 通过文件指针就可对它所指的文件进行各种操作。
定义说明文件指针的一般形式为:

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。



8.4 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	fscanf	按照格式化进行读写文件

8.5 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：

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

方式	含义
	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

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);
```

c 语言中有三个特殊的文件指针无需定义、打开可直接使用:

stdin： 标准输入 默认为当前终端（键盘）

我们使用的 scanf、getchar 函数默认从此终端获得数据

stdout：标准输出 默认为当前终端（屏幕）

我们使用的 printf、puts 函数默认输出信息到此终端

stderr：标准出错 默认为当前终端（屏幕）

当我们程序出错或者使用:perror 函数时信息打印在此终端

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
```

(3) 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。

```c
#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;
}
```

```c
#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;
}
```

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;
}
```

8.6 文件操作案例

（1）配置文件读写（自定义接口）

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）

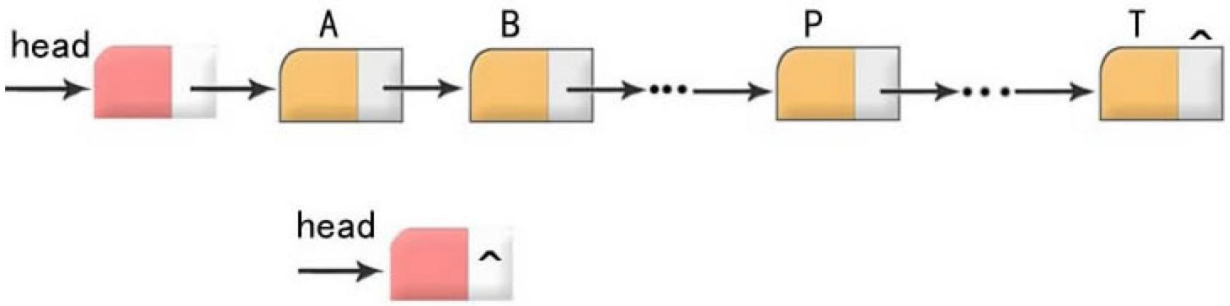
iv. 优化 ===》接口要求紧 模块要求松

2、 实现及代码讲解

3、 测试。

（2）加密文件读写 （使用别人写好的接口）

9. 链表和函数指针



- 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。

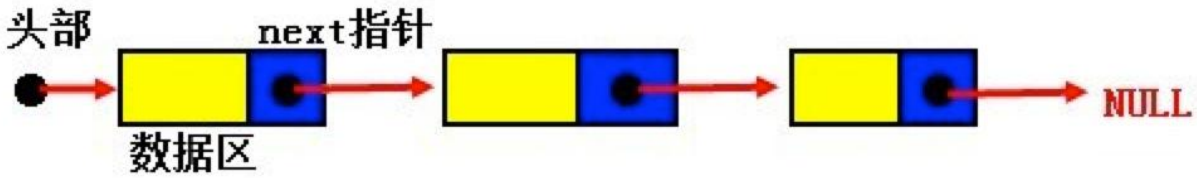
9.1 链表的相关概念

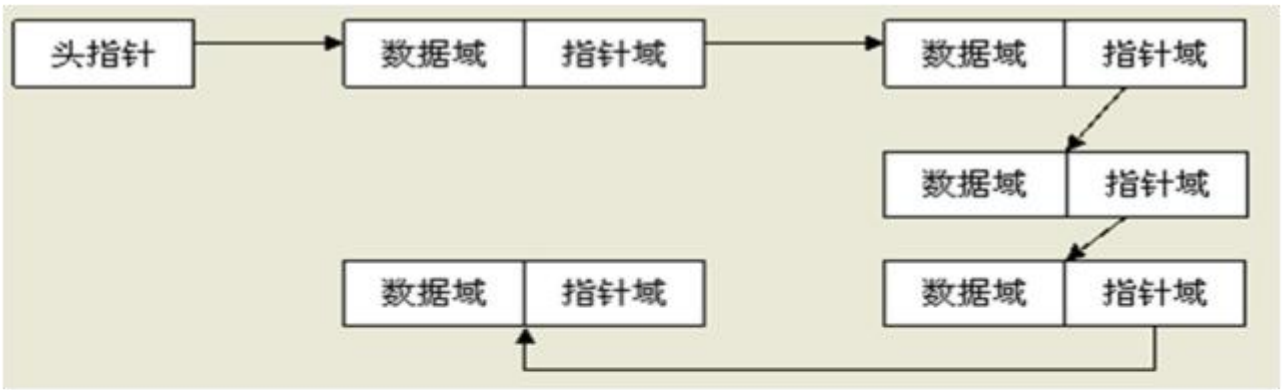
1）有关结构体的自身引用

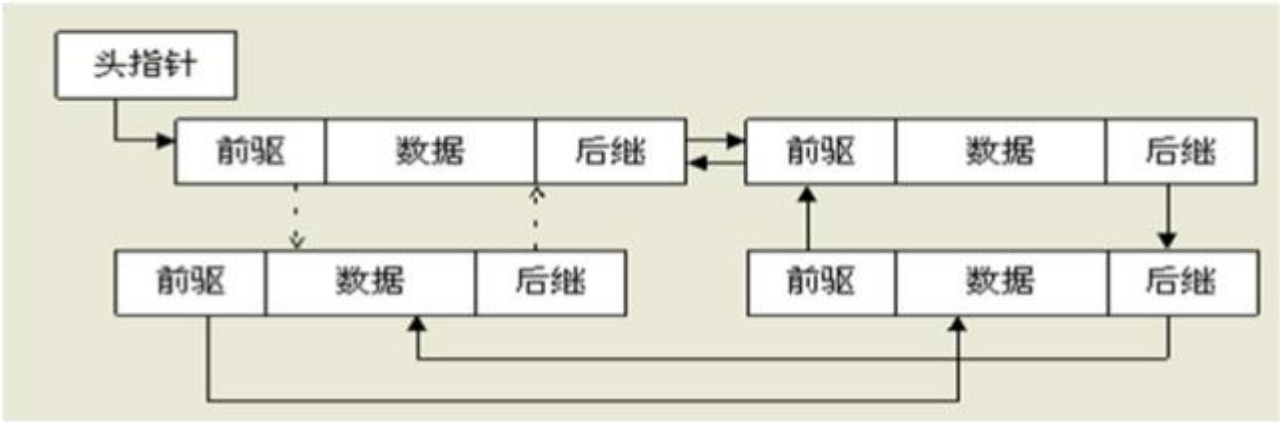
```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//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
};
```

9.2 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。

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”。

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。

2）带头链表和不带头链表

3）单向链表、双向链表、循环链表等

9.3 基本操作

(1)建立带有头结点的单向链表

编写函数 SList_Creat，建立带有头结点的单向链表。循环创建结点，结点数据域中的数值从键盘输入，以-1 作为输入结束标志。链表的头结点地址由函数值返回。

(2)顺序访问链表中各结点的数据域

编写函数 SList_Print，顺序输出单向链表各项结点数据域中的内容。

(3)在单向链表中插入节点

编写函数 SList_NodeInsert，功能：在值为 x 的结点前，插入值为 y 的结点；若值为 x 的结点不存在，则插在表尾。

(4)删除单向链表中的结点

编写函数 SList_NodeDel，删除值为 x 的结点。

9.4 指针函数和函数指针

(1) 指针函数（返回指针值的函数）

一个函数可以带回一个整型值、字符值、实型值等，也可以带回指针型的数据，即地址。

这种带回指针值的函数，一般定义形式为：

类型名 *函数名（参数表列）；

例如：

```
int *a(int x, int y);
```

(2) 函数指针（指向函数的指针）

一个函数在编译时被分配一个入口地址，这个地址就称为函数的指针，函数名代表函数的入口地址。

这一点和数组一样，因此我们可以用一个指针变量来存放这个入口地址，然后通过该指针变量调用函数。

```
int max (int x, int y);
int c;
c = max(a,b);
//这是通常用的方法，我们也可以定义一个函数指针，通过指针来调用这个函数。

int (*p)(int, int); //指向函数指针变量的定义形式
p = max;          //将函数的入口地址赋给函数指针变量 p
c = p(a,b);       //调用 max 函数
```

(3) 回调函数

函数指针变量常见的用途之一是把指针作为参数传递到其他函数，指向函数的指针也可以作为参数，以实现函数地址的传递。

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;
}
```

(4) 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;
}
```

- 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)

10. 预处理

10.1 预处理

1）预处理的基本概念

C 语言对源程序处理的四个步骤：预处理、编译、汇编、链接。

预处理是在程序源代码被编译之前，由预处理器（Preprocessor）对程序源代码进行的处理。这个过程并不对程序的源代码语法进行解析，但它会把源代码分割或处理成为特定的符号为下一步的编译做准备工作。

2）预编译命令

C编译器提供的预处理功能主要有以下四种：

1)文件包含 #include

2)宏定义 #define

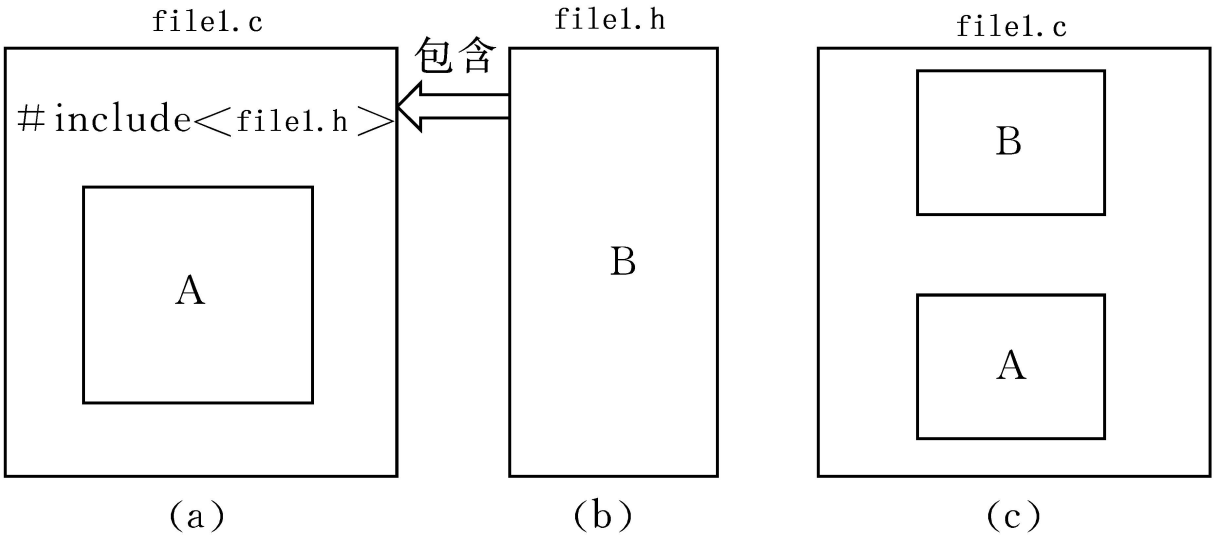
3)条件编译 #if #endif ..

4)一些特殊作用的预定义宏

10.2 文件包含处理

1）文件包含处理

文件包含处理”是指一个源文件可以将另外一个文件的全部内容包含进来。C语言提供了#include 命令用来实现“文件包含”的操作。



2）#include< > 与 #include ""的区别

""表示系统先在 file1.c 所在的当前目录找 file1.h，如果找不到，再按系统指定的目录检索。

< >表示系统直接按系统指定的目录检索。

注意：

1. #include <>常用于包含库函数的头文件
2. #include ""常用于包含自定义的头文件
3. 理论上#include 可以包含任意格式的文件(.c .h 等)，但我们一般用于头文件的包含。

10.3 宏定义

1）基本概念

在源程序中，允许一个标识符（宏名）来表示一个语言符号字符串用指定的符号代替指定的信息。

在 C 语言中，“宏”分为：无参数的宏和有参数的宏。

2）无参数的宏定义

#define 宏名 字符串

例: #define PI 3.141926

在编译预处理时，将程序中在该语句以后出现的所有的 PI 都用 3.1415926 代替。

这种方法使用户能以一个简单的名字代替一个长的字符串，在预编译时将宏名替换成字符串的过程称为“宏展开”。宏定义，只在宏定义的文件中起作用。

```
#include <stdio.h>

#define PI 3.1415f

int main(void)
{
    float L,S,R,V;
    printf("Input Radius:");
    scanf("%f",&R);

    L=2.0f*PI*R;
    S=PI*R*R;
    V=4.0f/3*PI*R*R*R;

    printf("L=%.4f,S=%.4f,V=%.4f\n",L,S,V);

    return 0;
}
```

说明:

1) 宏名一般用大写，以便于与变量区别

2) 字符串可以是常数、表达式等

3) 宏定义不作语法检查，只有在编译被宏展开后的源程序才会报错

4) 宏定义不是 C 语言，不在行末加分号

5) 宏名有效范围为从定义到本源文件结束

6) 可以用#undef 命令终止宏定义的作用域

7) 在宏定义中，可以引用已定义的宏名

3）带参数的宏定义

1) 格式：#define 宏名(形参表) 字符串

2) 调用：宏名(形参表)

3) 宏展开：进行宏替换

#define S(a,b) a*b

……

Area = S(3,2);

```
#include <stdio.h>

#define SQ_1(y) (y)*(y)
#define SQ_2(y)  y*y

int main(void)
{
    int a = 0,num_1 = 0,num_2 = 0;
    scanf("%d",&a);
    num_1 = SQ_1(a+1); //num_1 = (a+1)*(a+1);
    num_2 = SQ_2(a+1); //num_2 = a+1*a+1;

    printf("num_1 = %d\n",num_1);
    printf("num_2 = %d\n",num_2);

    return 0;
}
```

10.4 条件编译

1）基本概念

一般情况下，源程序中所有的行都参加编译。但有时希望对部分源程序行只在满足一定条件时才编译，即对这部分源程序行指定编译条件。

测试存在：

```
#ifdef 标识符
    程序段1
#else
    程序段2
#endif
```

测试不存在：

```
#ifndef 标识符
    程序段1
#else
    程序段2
#endif
```

根据表达式定义：

```
#if 表达式
    程序段1
#else
    程序段2
#endif
```

2）条件编译的作用

1）防止头文件被重复包含引用

```
#ifndef _SOMEFILE_H
#define _SOMEFILE_H

    //需要声明的变量、函数
    //宏定义
    //结构体

#endif
```

2) 软件裁剪

同样的 C 源代码，条件选项不同可以编译出不同的可执行程序。

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

#define BIG 1
int main(void)
{
    char str[20] = "C Language";
    char C;
    int i = 0;
    while ((C = str[i++]) != '\0')
    {
#if BIG
        if (C >= 'a' && C <= 'z')
            C = C - 32;
#else
        if (C >= 'A'&& C <= 'Z')
            C = C + 32;
#endif
```

```
        printf("%c", C);
    }

    system("pause");

    return 0;
}
```

10.5 一些特殊的预定宏

C 编译器，提供了几个特殊形式的预定义宏，在实际编程中可以直接使用，很方便。

```
//    __FILE__              宏所在文件的源文件名
//    __LINE__              宏所在行的行号
//    __DATE__              代码编译的日期
//    __TIME__              代码编译的时间

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

int main(void)
{

    printf("%s\n", __FILE__);
    printf("%d\n", __LINE__);
    printf("%s\n", __DATE__);
    printf("%s\n", __TIME__);

    system("pause");

    return 0;
}
```

11. 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。

xxx.lib：编译代码时，需要链接此文件

xxx.dll：运行程序时，需要链接此文件

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头衔。

__declspec(dllexport)

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>
#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：

```
#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 输出結果

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。