传智播客 C 提高讲义

传智扫地僧

1 程序内存模型

1.1 就业班引言

1.1.1 问题引出

企业需要能干活的人

- C 学到什么程度可以找工作？
- 对于 C/C++初级开发者，怎么达到企业的用人标准
- 就业问题

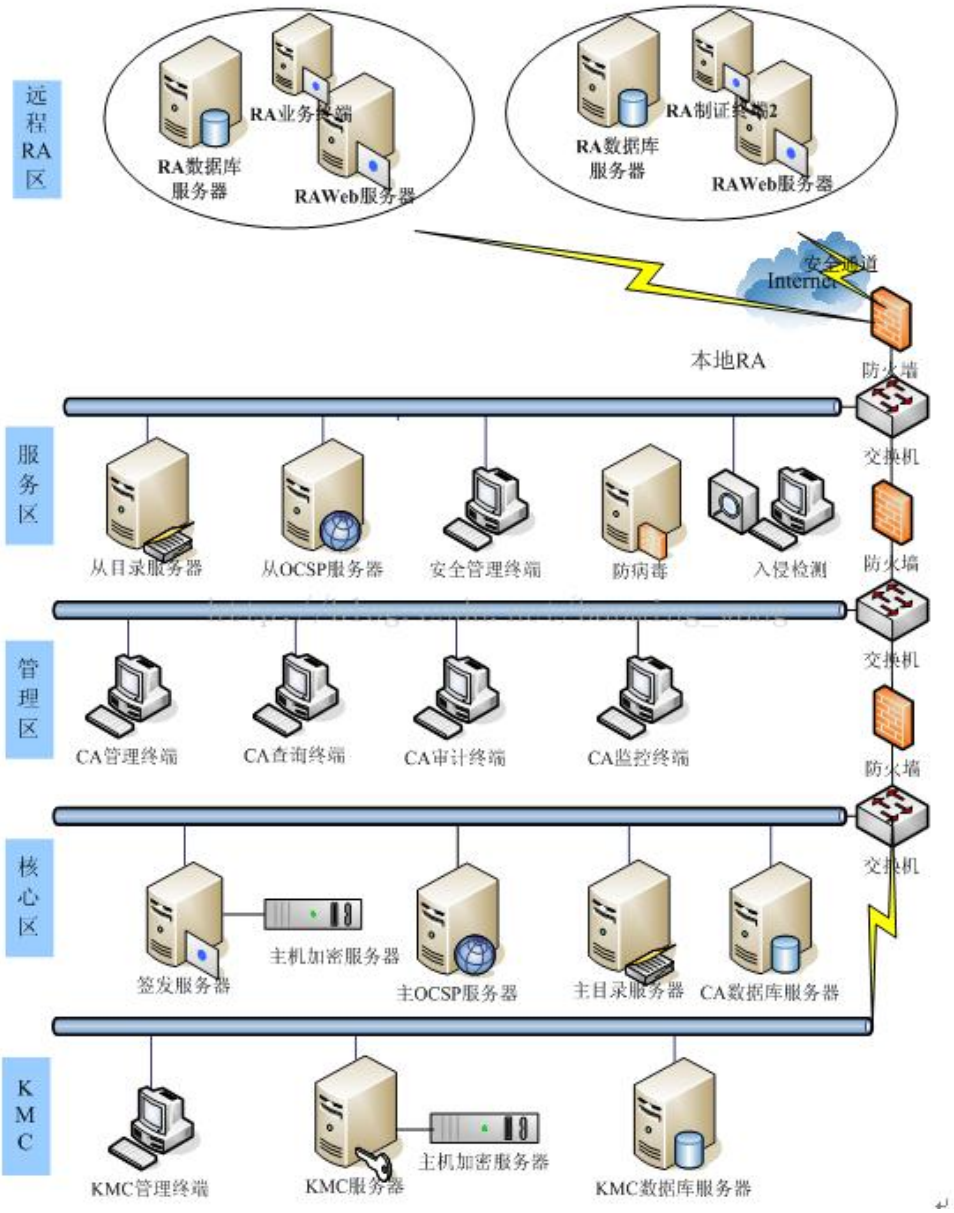
问： 老师，有没有一个框框？

有没有一个标准啊？

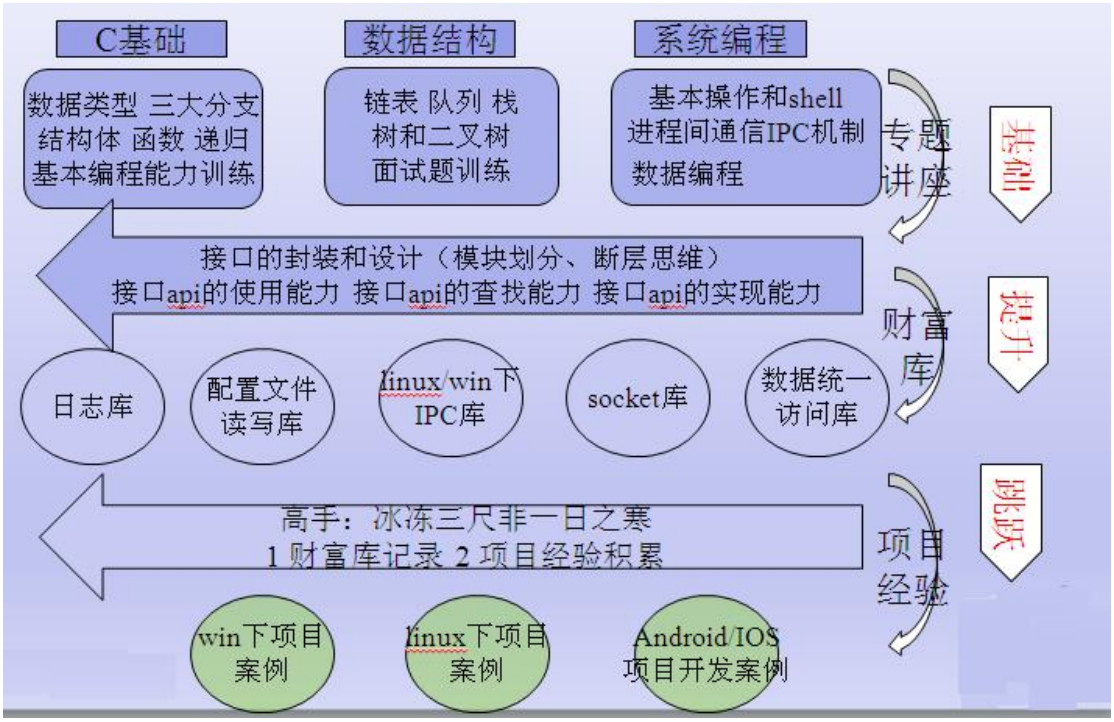
我们学什么哪？

C 工程开发需要什么（培养什么能力）

成熟的、商业化的信息系统在分区、分层



信息系统的技术模型在分层



找出对我们初学者最近的那一层（哪些能力是你入行前，必须要掌握的）

C 项目开发的套路（一套接口）

- //socket_client pool api 设计与实现
- int sckClient_poolinit(void **handle);
- int sckClient_getConnet(void *handle, void **hConnect);
- int sckClient_sendData(void *hConnect, unsigned char *data, int dataLen);
- int sckClient_getData(void *hConnect, unsigned char **data, int *dataLen);
- int sckClient_getData_Free(void *hConnect, unsigned char *data);
- int sckClient_putConnet(void *handle, void **hConnect);
- int sckClient_pooldestory(void **handle);

总结：寻找到学习的标准

培养两种能力

- 接口的封装和设计（功能抽象和封装）
  - 接口 api 的使用能力
  - 接口 api 的查找能力（快速上手）

- 接口 api 的实现能力

- 建立正确程序运行内存布局图（印象图）
  - 内存四区模型图
  - 函数调用模型图

1.1.2 总体课程安排

课程大纲

- C 提高
- C++
- 数据结构
- 总体时间 1 个月

实用专题

- 总：轻松入门 实战应用
- 形式 1：专题的形式录制 话题集中 便于初学者学习
- 形式 2：知识点分段录制、细致讲解，从根本上提高初学者水平
- 项目开发中的重要点做剖析
  - 指针铁律 1 2 3 4 5 6 7 8 9 10===》企业用人标准

1.1.3 学员要求

- 资料，时间空间管理
- 工作经验，记录和积累
- 临界点
  - 事物认知规律
  - 挑战 *p，**p, ***p
- 提高课堂效率
  - 课堂例子，当堂运行。
  - 录制视频说明（不来，看视频）

- C/C++学习特点
  - Java：学习、应用、做项目
  - C：学习、理解、应用、做项目
- 多动手
  - 不动手，永远学不会
  - 关键点、关键时候，进行强化训练和考试

1.1.4 小结

- 建立信心
  - 接口的封装和设计
  - 指针教学，多年实践检验
- 心态放轻松了
  - 分析有效时间
  - 尊重事物认知规律、给自己一次机会

1.2 学员听课的标准

C 语言学到什么程度，就可以听懂传智播客就业班第一阶段的课程了。

有没有一个标准？

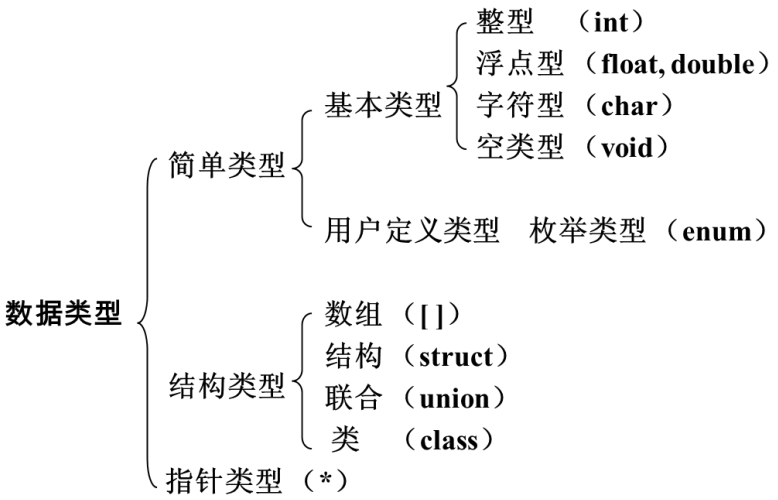
- 选择法或者冒泡法排序
  - 在一个函数内排序
  - 通过函数调用的方式排序
  - 数组做函数参数的技术盲点和推演

1.3 内存四区模型

1.3.1 数据类型本质分析

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？

数据类型的本质

- 数据类型可理解为创建变量的模具（模子）；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
- 程序举例，如何求数据类型的大小 sizeof(int *)
- 请问：数据类型可以有别名吗？数据类型可以自定义吗？

数据类型大小

```
int main()
{
int a = 10;
int b[10] ;
printf("int a:%d \n", sizeof(a));
printf("int a:%d \n", sizeof(int *));
printf("int b:%d \n", sizeof(b));
printf("int b:%d \n", sizeof(b[0]));
printf("int b:%d \n", sizeof(*b));
printf("hello.....\n");
getchar();
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

数据类型别名

- 数据类型可以理解为固定大小内存块的别名，请问数据类型可以起别名吗？

```
int main()
{
	//Teacher t1;
	printf("Teacher:%d \n", sizeof(Teacher));
	printf("u32:%d \n", sizeof(u32));
	printf("u8:%d \n", sizeof(u8));
	printf("hello.....\n");
	getchar();
	return 0;
}
```

数据类型的封装

- 1、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。
- 2、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 );
```

- 3、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;}
```

- 4、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);
```

- 5、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

- 6、扩展阅读《void 类型详解.doc》

数据类型总结与扩展

- 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 一维数组、二维数组有数据类型吗？int array[10]。

若有，数组类型又如何表达？又如定义？

若没有，也请说明原因。

抛砖：数组类型，压死初学者的三座大山

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。

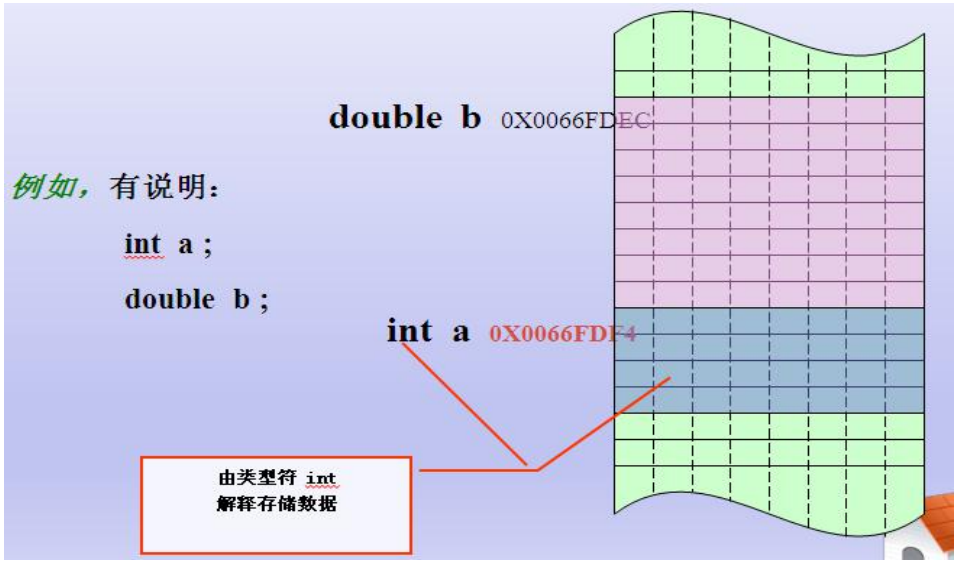
抛砖：

1.3.2 变量本质分析

变量概念

- 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 ；
- *例如：*

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



变量本质

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

（一段连续）内存空间的别名（是一个门牌号）

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；于是横空出世了！（编程案例）

3、内存空间可以再取给别名吗？

4、数据类型和变量的关系

- 通过数据类型定义变量

5、总结及思考题

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。问：变量跑哪去了？

思考 1：变量三要素(名称、大小、作用域)，变量的生命周期？

思考 2：C++编译器是如何管理函数 1，函数 2 变量之间的关系的？

====》引出两个重要话题：

内存四区模型

函数调用模型

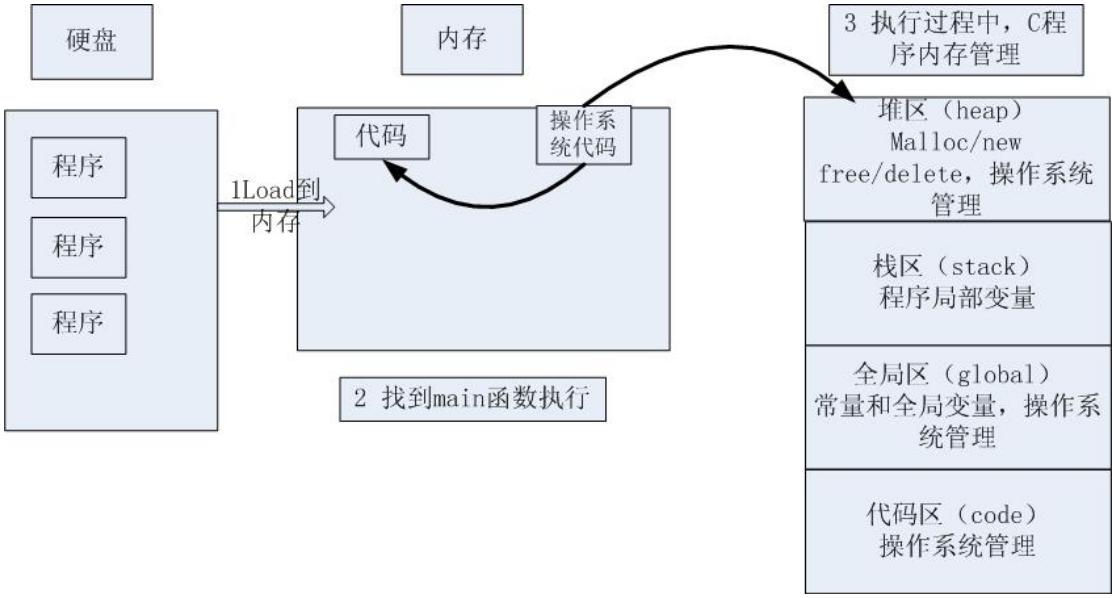
重要实验：

```
int  main333()
{
    //
    //2 种方法，通过变量直接操作内存
    //   通过内存编号操作内存

    int i = 0;

    printf("&i:%d\n", &i);

    *((int *)(1245024)) = 10;
    printf("i:%d", i);
```

```
    printf("hello....\n");
    getchar();
    return 0;
}
```

1.3.3 程序的内存四区模型

内存四区的建立流程



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

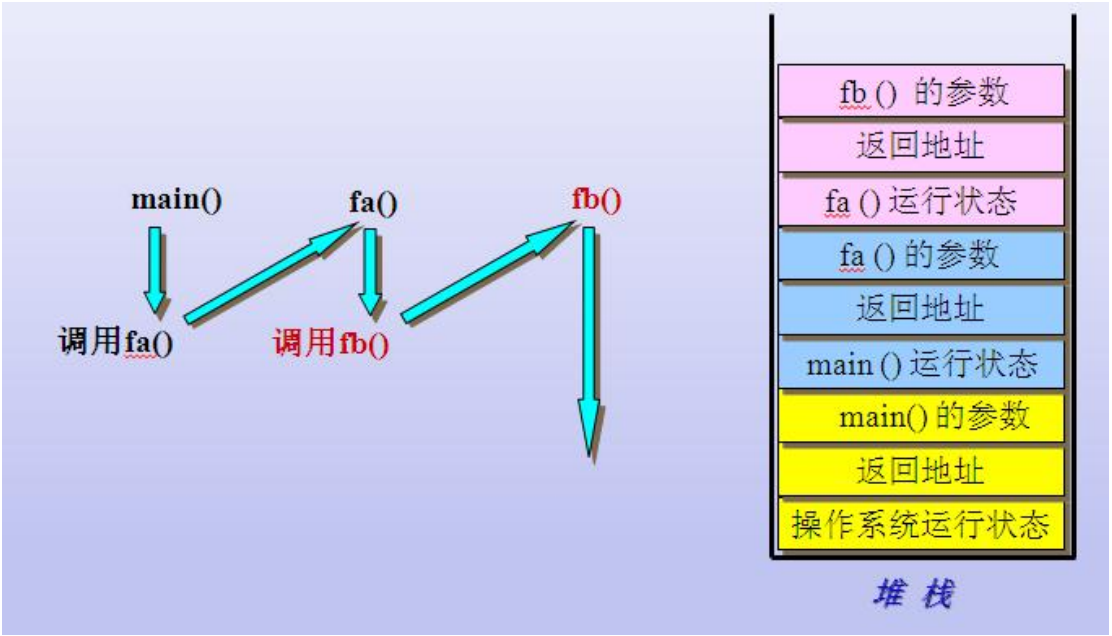
3、操作系统找到 main 函数入口执行

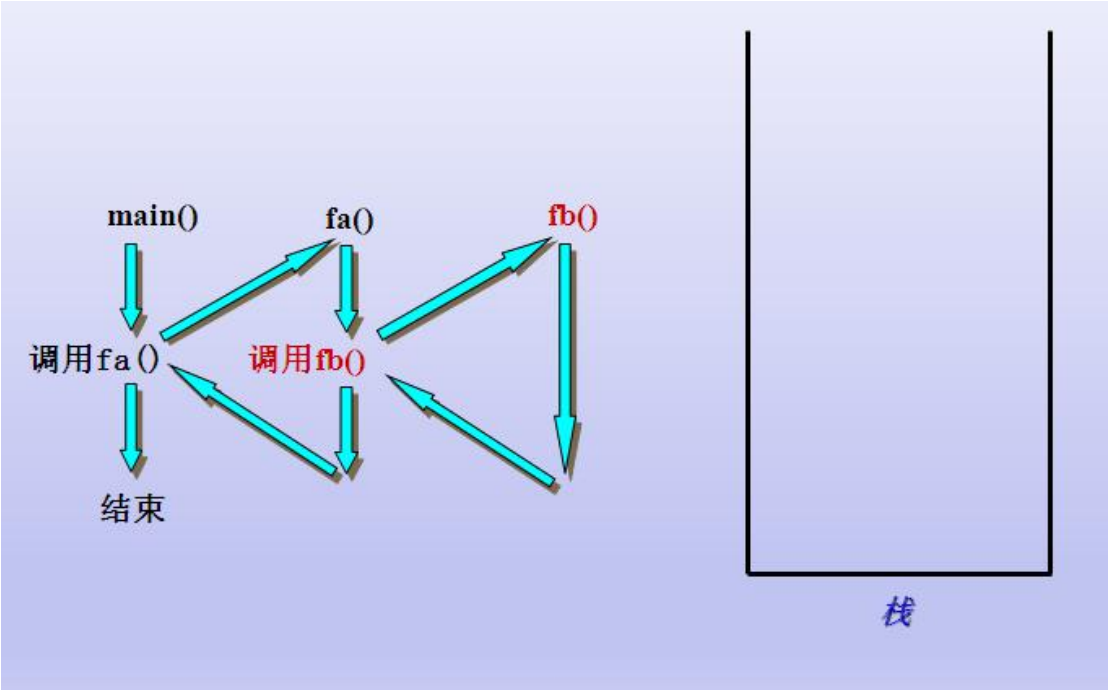
各区元素分析

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。

1.4 函数调用模型

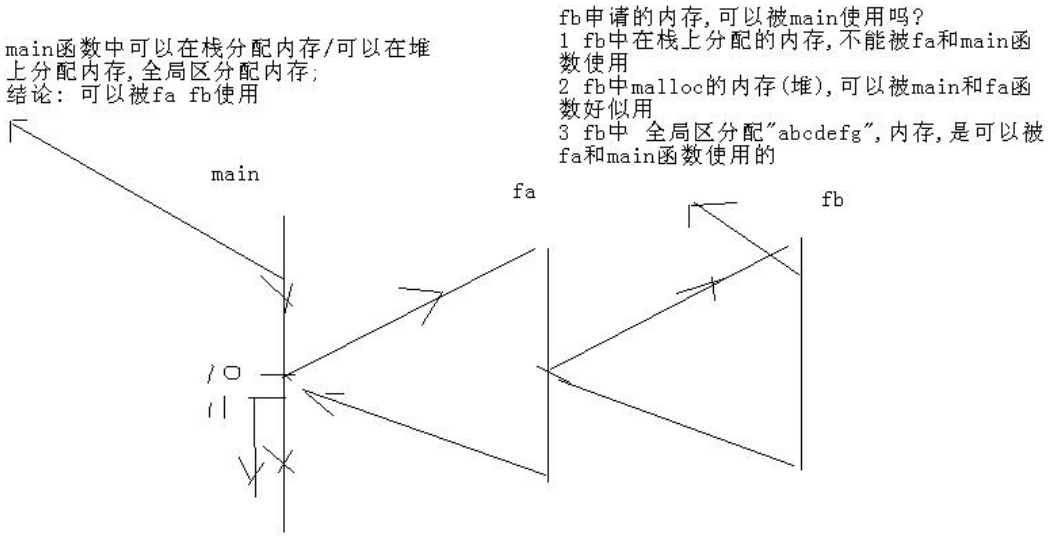
1.4.1 基本原理





1.4.2 内存四区模型和函数调用模型变量传递分析

1、一个主程序有 n 函数组成，c++编译器会建立有几个堆区？有几个栈区？
2、函数嵌套调用时，实参地址传给形参后，C++编译器如何管理变量的生命周期？
分析：函数 A，调用函数 B，通过参数传递的变量（内存空间能用吗？）



1.4.3 提示学好 C 语言的关键

1.4.4 如何建立正确的程序运行内存布局图

- 内存四区模型&函数调用模型
- 函数内元素
  - 深入理解数据类型和变量“内存”属性
  - 一级指针内存布局图(int *,char*)
  - 二级指针内存布局图(int ** char **)
- 函数间
  - 主调函数分配内存，还是被调用函数分配内存
  - 主调函数如何使用被调用函数分配的内存（技术关键点：指针做函数参数）

======》学习指针的技术路线图

1.5 内存四区强化训练

01 全局区训练

- char *p1= “abcdefg”;

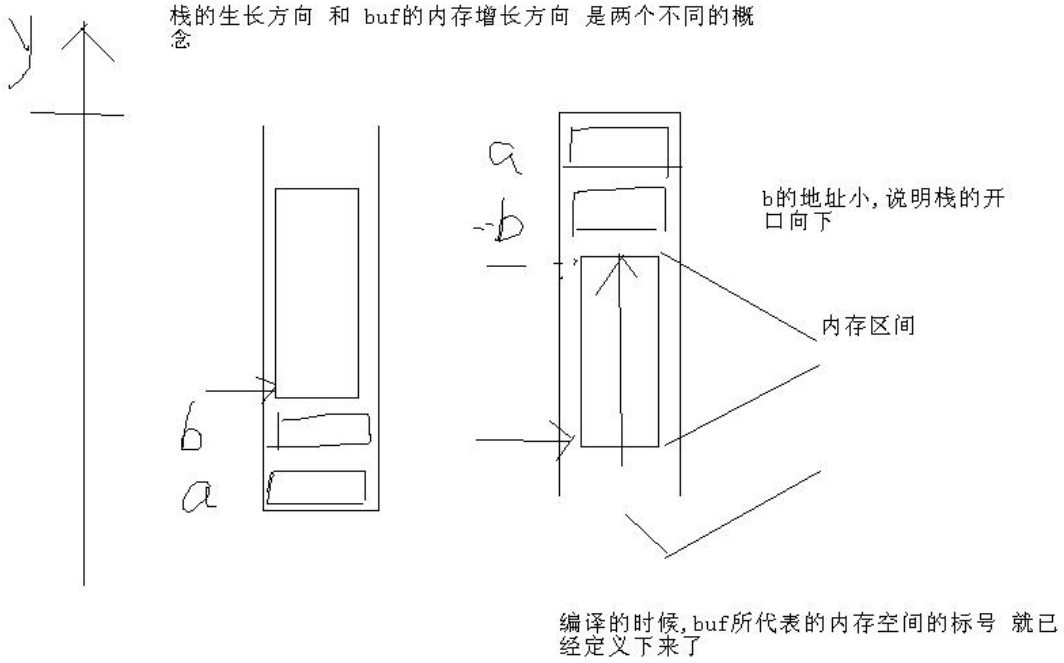
02 堆栈区生命周期训练

- Char p1[]= “abcdefg”;
- 返回基本类型

- 返回非基本类型

03 堆栈属性训练

- 测试 heap 生长方向
- 测试 stack 生长方向
  - Heap、stack 生长方向和内存存放方向是两个不同概念
  - 野指针
  - Malloc 得到指针释放问题测试
    - free(p)
    - free(p+1)，深入理解



1.6 作业强化

训练 1 划出内存四区

```
void main26()
{
    char buf[100];
    //byte b1 = new byte[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 *p = NULL; //分配 4 个字节的内存 栈区也叫临时区
        p = (char *)malloc(100); //内存泄露概念
        if (p != NULL)
        {
            free(p);
        }
    }

    system("pause");
}
```

全局区代码测试

```
char * getstring1()
{
    char *p1 = "abcde";
    return p1;
}

char * getstring2()
{
    char *p2 = "abcde";
    return p2;
```

```
}



void main()
{
	int i= 0;

	//指针指向谁就把谁的地址赋给指针变量。
	char *p1 = getstring1();
	char *p2 = getstring2();
	char *******	p3 = NULL; //p3  是个变量

	//指针变量和它所执行的内存空间变量是两个不同的概念
	strcmp(p1, p2);

	system("pause");
}
```

训练 2 划出内存四区

```
void main01()
{
	char buf[100];
	//byte b1 = new byte[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 *p2 = NULL; //分配 4 个字节的内存  栈区也叫临时区
        p2 = (char *)malloc(100); //内存泄露概念
        if (p2 != NULL)
        {
            free(p2);
            //p2 = NULL;    若不写,实验效果,分析原因
        }
        if (p2 != NULL)
        {
            free(p2);
        }
    }

    system("pause");
}
```

2 指针知识体系搭建

2.1 前言

两个模型（内存四区模型和函数调用模型）
先从整体上把握指针的知识体系。然后突破 1 级指针、二级指针、多级指针。

2.2 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

//含义 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++

//含义 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

//含义 3 =左边*p 表示 给内存赋值， =右边*p 表示取值 含义不同切结！

//含义 4 =左边 char *p

//含义 5 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

含义 1：指针步长（p++），根据所致内存空间的数据类型来确定

p++=➔(unsigned char)p+sizeof(a);

结论：指针的步长，根据所指内存空间类型来定。

注意： 建立指针指向谁，就把把谁的地址赋值给指针。图和代码和二为一。

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）。

强化 2：间接赋值（*p）是指针存在的最大意义

1）两码事：指针变量和它指向的内存块变量

2）条件反射：指针指向某个变量，就是把某个变量地址否给指针

3）*p 间接赋值成立条件：3 个条件

a)2 个变量（通常一个实参，一个形参）

b) 建立关系，实参取地址赋给形参指针

c) *p 形参去间接修改实参的值

Int iNum = 0; //实参
int *p = NULL;
p = &iNum;
iNum = 1;
*p =2 ; //通过*形参 == 间接地改变实参的值
*p 成立的三个条件：

4）引申： 函数调用时,用 n 指针（形参）改变 n-1 指针（实参）的值。

//改变 0 级指针（int iNum = 1）的值有 2 种方式
//改变 1 级指针（eg char *p = 0x1111 ）的值，有 2 种方式
//改变 2 级指针的（eg char **pp1 = 0x1111 ）的值，有 2 种方式

//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。
//指针作为函数参数的精髓。

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。

强化 4：应用指针必须和函数调用相结合（指针做函数参数）

编号	指针函数参数 内存分配方式（级别+堆栈）		主调函数 实参	被调函数 形参	备注
01	1 级指针 （做输入）	堆	分配	使用	一般应用禁用
		栈	分配	使用	常用
		Int showbuf(char *p); int showArray(int *array, int iNum)			
02	1 级指针 （做输出）	栈	使用	结果传出	常用
		int geLen(char *pFileName, int *pfileLen);			
03	2 级指针 （做输入）	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			
04	2 级指针 （做输出）	堆	使用	分配	常用，但不建议用，转化成 02

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

05	3 级指针 （做输出）	堆	使用	分配	不常用
		int getFileAllLine(char ***content, int *pLine); int getFileAllLine_Free(char ***content, int *pLine);			

指针做函数参数，问题的实质不是指针，而是看内存块，内存块是 1 维、2 维。
1）如果基础类 int 变量，不需要用指针；
2）若内存块是 1 维、2 维。

强化 5：一级指针典型用法（指针做函数参数）

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 geLen(char *pFileName, int *pfileLen);

理解

主调函数还是被调用函数分配内存

被调用函数是在 heap/stack 上分配内存

强化 6：二级指针典型用法（指针做函数参数）

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

理解

主调函数还是被调用函数分配内存

被调用函数是在 heap/stack 上分配内存

强化 7： 三级指针输出典型用法

三级指针做输出
int getFileAllLine(char ***content, int *pLine);
int getFileAllLine_Free(char ***content, int *pLine);
理解
主调函数还是被调用函数分配内存
被调用函数是在 heap/stack 上分配内存

强化 8：杂项，指针用法几点扩充

1）野指针 2 种 free 形式
int getData(char **data, int *dataLen);
int getData_Free(void *data);
int getData_Free2(void **data);
2）2 次调用
主调函数第一次调用被调用函数求长度；根据长度，分配内存，调用被调用函数。
3）返回值 char */int/char **
4）C 程序书写结构
商业软件，每一个出错的地方都要有日志，日志级别

强化 9：一般应用禁用 malloc/new

2.3 接口封装设计思想引导

基于 socketclient 客户端接口设计与实现（仿真模拟）

2.4 附录

【王保明老师经典语录】

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）指针指向谁就把谁的地址赋给指针，用它对付链表轻松加愉快

7）链表入门的关键是分清楚链表操作和辅助指针变量之间的逻辑关系

8）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。多了一个步骤吧。

9) 学好指针才学会了 C 语言的半壁江山，另外半壁江山在哪里呢？你猜，精彩剖析在课堂。

10) 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，哪来的指针啊。

3 字符串和一级指针内存模型专题

3.1 字符串基本操作

字符数组初始化方法

```
int main11()
{
	//1 大{}号法 初始化列表
	//数组初始化有 2 种方法 默认元素个数、指定元素个数
	char buf1[] = {'a', 'b', 'c', 'd', 'e'};  //若没有指定长度，默认不分配零
	//若指定长度，不够报错；buf 长度多于初始化个数，会自动补充零
	char buf2[6] = {'a', 'b', 'c', 'd', 'e'};
	char buf3[6] = {'a', 'b', 'c', 'd', 'e'};
	//char buf4[5] = {'a', 'b', 'c', 'd', 'e'};
	printf("buf3:%s", buf3);
```

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

在 C 语言中使用字符数组来模拟字符串
C 语言中的字符串是以’\0’结束的字符数组
C 语言中的字符串可以分配于栈空间，堆空间或者只读存储区

```
//在 C 语言中使用字符数组来模拟字符串
//C 语言中的字符串是以’\0’结束的字符数组
//C 语言中的字符串可以分配于栈空间，堆空间或者只读存储区
int main12()
{
    //1 用字符串来初始化数组
    char buf2[] = {'a', 'b','c','d','\0'};
    //2 字符串常量初始化一个字符数组
    char buf3[] = {"abcde"}; //结论：会补充零
    char buf4[] = "abcde";
    char buf5[100] = "abcde";

    printf(" strlen(buf5) :%d \n", strlen(buf5));
    printf(" sizeof(buf4) :%d \n", sizeof(buf5));
    printf(" sizeof(buf4) :%d \n", sizeof(buf4));
}
```

```
//strlen()求字符串的长度，注意字符串的长度不包含\0
//sizeof(类型)字符串类型，的大小，包括\0；
```

02Sizeof 与 strlen 的区别

数组法和指针法操作字符串

03 字符串操作
数组法，下标法
字符数组名，是个指针，是个常量指针；
字符数组名，代表字符数组首元素的地址，不代表整个数组的。
如果代表这个数组，那需要数组数据类型的知识！
下期分解

```
//字符串操作方法 数组下标法 指针法
int main13()
{
    int i = 0;
    char buf5[100] = "abcde";
    char *p = NULL;

    //下标法
    for (i=0; i<100; i++)
```

```
    {
        printf("%c", buf5[i]);
    }
    printf("\n");

    //指针法 1
    for (i=0; i<100; i++)
    {
        printf("%c", *(buf5+i));
    }
    //buf5 是个指针，是个常量指针

    //指针法 2
    printf("\n");
    p = buf5;
    for (i=0; i<100; i++)
    {
        printf("%c", *(p+i));
    }
    //buf5 是个指针，是个常量指针
}
```

推演过程为：i 变 0+I，去[]号加*号
//其实本质：指针*p 间接寻址，操作内存；
//[] 编译器为我们做了*p 操作而已

3.2 字符串做函数参数

深入理解指针。。。。。。。。。。。的关键是什么？
注意
指针和数组的巨大区别
char *p = “abcdefg”;
Char *buf = “abcdefg”;
一维字符串内存模型：两种

```
void copy_str01(char *from, char *to)
{
    for (; *from!='\0'; from++, to++)
    {
        *to = *from;
    }
    *to = '\0';
```

```
}

void copy_str02(char *from, char *to)
{
	while(*from!='\0')
	{
		*to++ = *from++;
	}
	*to = '\0';
}

void copy_str03(char *from, char *to)
{
	while( (*to=*from) !='\0')
	{
		to++;
		from++;
	}
}

void copy_str04(char *from, char *to)
{
	while( (*to++=*from++) !='\0')
	{
		;
	}
}

int copy_str05_good(const char *from, char *to)
{
	if (from==NULL || to==NULL)
	{
		printf("func copy_str05_good() err. (from==NULL || to==NULL)\n");
		return -1;
	}

	while( (*to++=*from++) !='\0')
	{
		;
	}
	return 0;
}
```

典型错误知多少

```
char *str_cnct(char *x, char* y)		/*简化算法*/
```

```
{
	char str3[80];
	char *z=str3;  /*指针 z 指向数组 str3*/
	while(*z++=*x++);
	z--;                    /*去掉串尾结束标志*/
	while(*z++=*y++);
	z=str3;          /*将 str3 地址赋给指针变量 z*/
	return(z);
}
```

修改字符常量结果会如何

```
Char *p = “abcdefg”;
Modify p[1] = ‘1’;
```

04 字符串操作易错

```
//你往哪里输入数据
int main()
{
	char buf[2000];
	char *p = NULL;
	p    = buf;
	printf("\n 请输入一个字符串:");
	scanf("%s", p);
	printf("%s", p);

	getchar();
	getchar();
	return 0;
}
```

3.3 库函数 api

快速的上手 api 是一种能力!

建立正确的程序运行示意图，（内存四区及函数调用堆栈图）是根本保障!!

```
int main31()
{
	char buf1[100];
	char buf2[200];
	strcpy(buf1, "111");
	printf("%s", strcat(buf1, "222"));
	getchar();
	return 0;
}

int main32()
```

```
{
	char *string1 = "1234567890";
	char *string2 = "747DC8";
	int length;
	//在字符 str1 中查找，与 str2 中任意字符有公共交集的位置
	length = strcspn(string1, string2);
	printf("Character where strings intersect is at position %d\n", length);

	getchar();
	return 0;
}

//strnset 函数有错误
//测试程序修改如下
int main33()
{
	char string[] = "abcdefghijklmnopqrstuvwxyz";
	char letter = 'x';
	printf("string before strnset: %s\n", string);
	strnset(string, letter, 13);
	printf("string after    strnset: %s\n", string);
	getchar();
	return 0;
}

int main44()
{
	char *string1 = "abcdefghijklmnopqrstuvwxyz";
	char *string2 = "onm";
	char *ptr;
	ptr = strpbrk(string1, string2);
	if (ptr)
		printf("strpbrk found first character: %c\n", *ptr);
	else
		printf("strpbrk didn't find character in set\n");
	getchar();
	return 0;

}

int main55()
{
	 char input[16] = "abc,d";
    char *p;
```

```
    /* strtok places a NULL terminator
    in front of the token, if found */
    p = strtok(input, ",");
    if (p)      printf("%s\n", p);
    /* A second call to strtok using a NULL
    as the first parameter returns a pointer
    to the character following the token    */
    p = strtok(NULL, ",");
    if (p)      printf("%s\n", p);

     getchar();
     return 0;

}

//典型的状态函数
int main()
{
      char str[] = "now # is the time for all # good men to come to the # aid of their country";
      //char delims[] = "#";
      char *delims = "#";
      char *result = NULL;
      result = strtok( str, delims );
      while( result != NULL ) {
            printf( "result is \"%s\"\n", result );
            result = strtok( NULL, delims );
      }
      printf("----------==========----------\n");
      printf("%s", str);

      getchar();

      return 0;
}
```

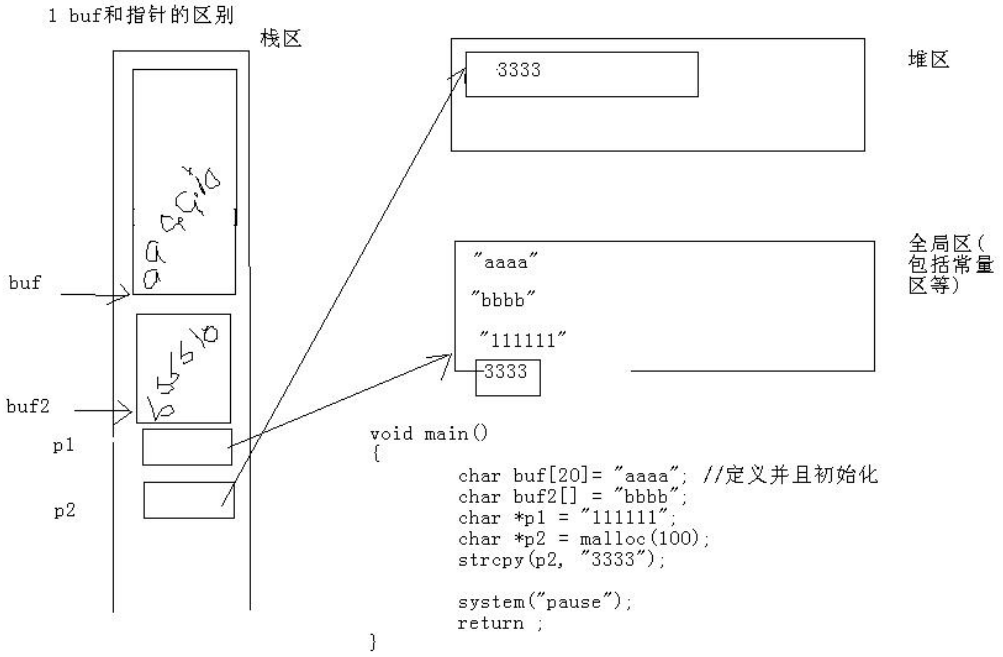
3.4 字符串相关一级指针内存模型

```
void main()
{
    char buf[20]= "aaaa";
    char buf2[] = "bbbb";
```

```
    char *p1 = "111111";
    char *p2 = malloc(100); strcpy(p2, "3333");

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

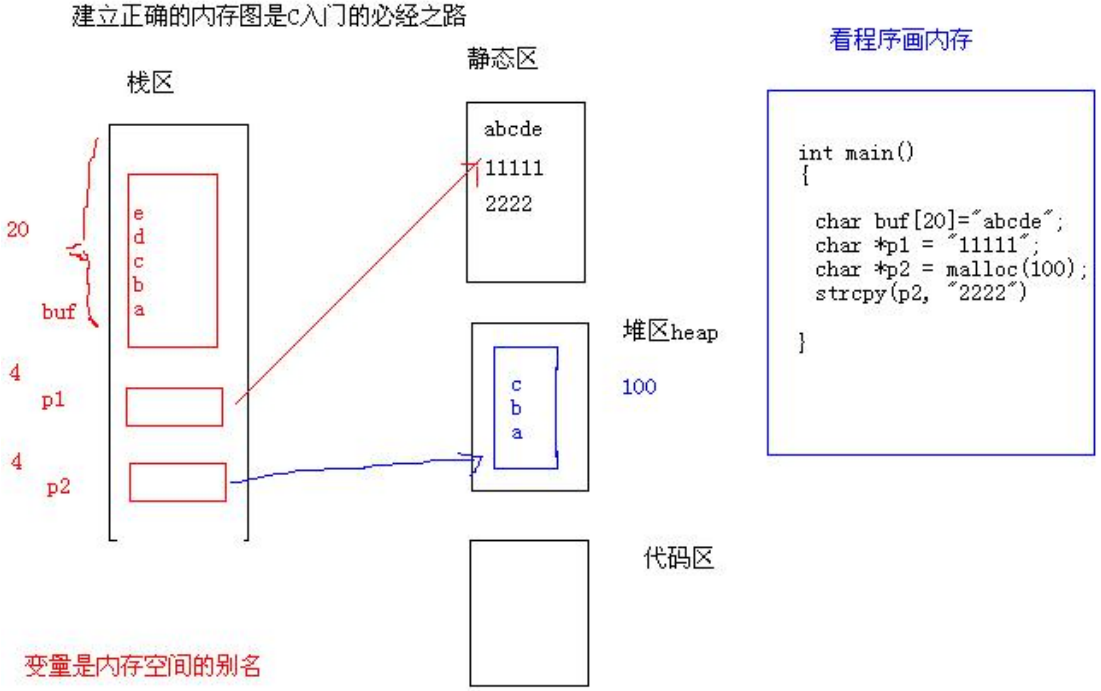
字符串1级指针的内存模型图

1 buf和指针的区别



一级指针（char *）内存印象图

建立正确的内存图是C入门的必经之路

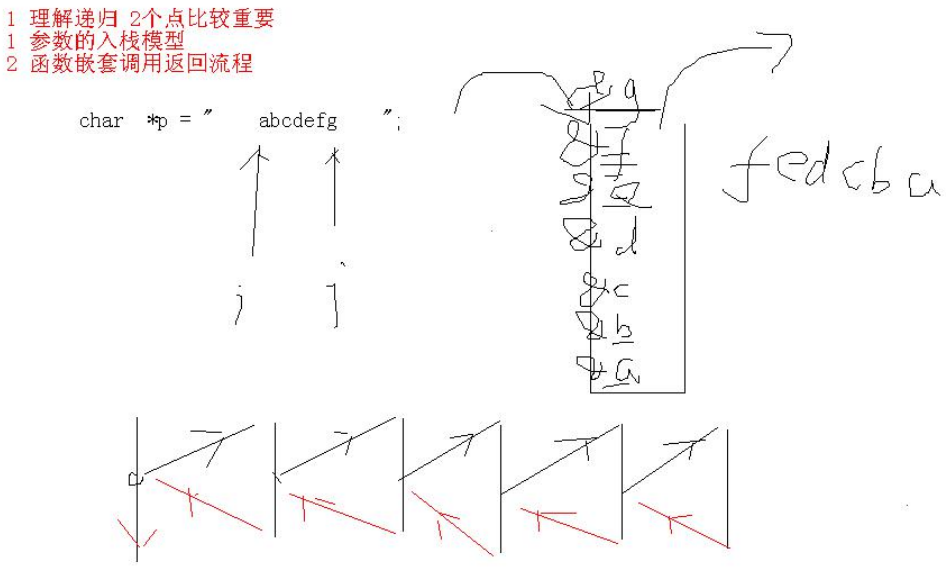


3.5 项目开发字符串经典案例

案例 1：strstr-whiledowhile 模型

案例 2：两头堵模型

案例 3：字符串反转模型



3.6 一级指针(char *)易错模型分析

01char *（字符串）做函数参数出错模型分析

建立一个思想：是主调函数分配内存，还是被调用函数分配内存；
//不要相信，主调函数给你传的内存空间，你可以写。。。。。。一级指针你懂了。 但是二级指针，你就不一定懂。。。抛出。。。。。。。。。

```
void copy_str21(char *from, char *to)
{

	if (*NULL = '\0' || *to!='\0')
	{
		Printf("func copy_str21() err\n");
		return;
	}


	for (; *from!='\0'; from++, to++)
	{
		*to = *from;
	}
	*to = '\0';
}
//字符串逆序
int main()
{
	//char p[1024] ={0};
char *p	={0}; p = NULL;

	char to[100];
	copy_str21(p, to);
```

C 语言中没有你不知道的，只有你不会调
Java 语言中没有你不会调的，只有你不知道

不断修改内存指针变量

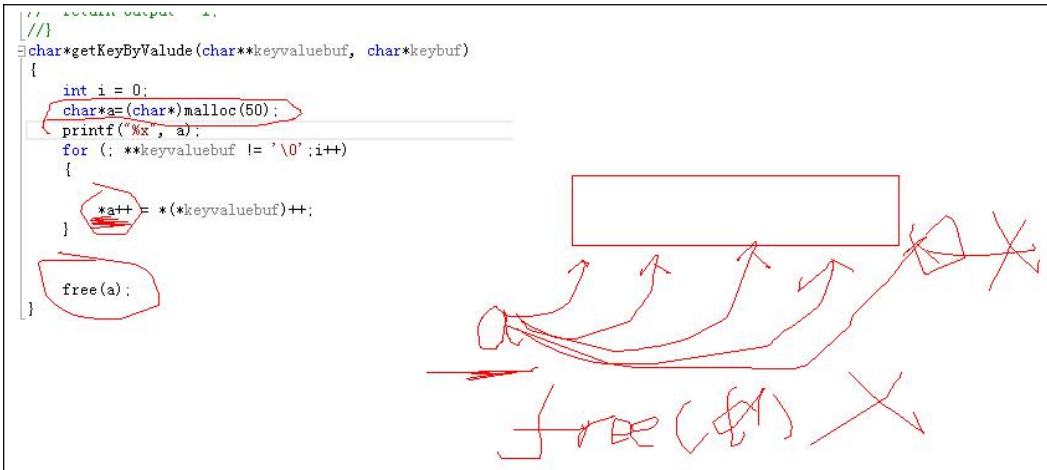
02 越界

越界 语法级别的越界

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

03 不断修改指针变量的值

越界




```
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);
}
```

04 你向外面传递什么

1、临时 str3 内存空间

```
// char *str_cnct(x,y)      /*简化算法*/
//  char *x,*y;
char *str_cnct(char *x, char* y)      /*简化算法*/
{
    char str3[80];
    char *z=str3;  /*指针 z 指向数组 str3*/
    while(*z++=*x++);
    z--;                    /*去掉串尾结束标志*/
```

```
    while(*z++=*y++);
    z=str3;      /*将 str3 地址赋给指针变量 z*/
    return(z);
}
2、经验要学习
while(*z++=*x++);
    z--;                    /*去掉串尾结束标志*/
```

```
char *str_cnct(char *x, char* y)      /*简化算法*/
{
    char * str3= (char *)malloc(80)
    char *z=str3;  /*指针 z 指向数组 str3*/
    while(*z++=*x++);
    z--;                    /*去掉串尾结束标志*/
    while(*z++=*y++);
    z=str3;      /*将 str3 地址赋给指针变量 z*/
    return(z);
}
```

```
char *str_cnct(char *x, char* y)      /*简化算法*/
{
If (x == NULL)
{
Return NULL;
}
    char * str3= (char *)malloc(80)
    char *z=str3;  /*指针 z 指向数组 str3*/
    while(*z++=*x++);
    z--;                    /*去掉串尾结束标志*/
    while(*z++=*y++);
    z=str3;      /*将 str3 地址赋给指针变量 z*/ note:
    return(z);

}
Main ()
{
Char *p = str_cnct(“abcd”, “ddeee”);
 If (p != NULL) {Free(p) ;p = NULL}//yezhizhen
}
```

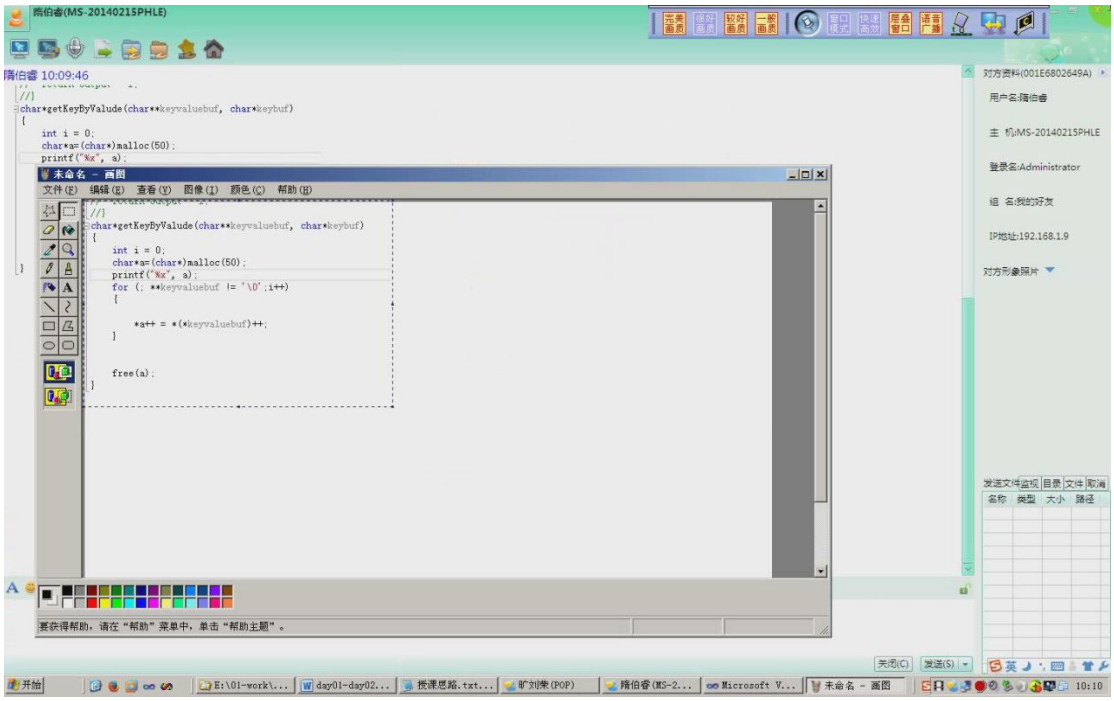
```
int getKeyByValude(char *keyvaluebuf,  char *keybuf,  char *valuebuf, int *
valuebuflen)
{
    int result = 0;
    char *getbuf = new char[100];
```

```
    memset(getbuf, 0, sizeof(getbuf));

    char *trimbuf = new char[100];
    memset(trimbuf, 0, sizeof(trimbuf));

    int destlen = strlen(keyvaluebuf);

    if (keybuf == NULL || keyvaluebuf == NULL || valuebuf == NULL/* || valuebuflen
== NULL*/)
    {
        result = -1;
        return  result;
    }

    if (strstr(keyvaluebuf, keybuf) == NULL)
    {
        result = -1;
        return result;
    }
    else
    {
        for (int i = 0; i < destlen; i++)
        {
            if (*keyvaluebuf == '=')
            {
                *keyvaluebuf++;
                break;
            }
            keyvaluebuf++;
        }
        while(*keyvaluebuf != '\0')
        {
            *valuebuf = *keyvaluebuf;
            valuebuf++;
            keyvaluebuf++;
        }
        *valuebuf = '\0';
    }

    int len = strlen(valuebuf);
    return result;
}
```

```
//char *p = "abcd11111abcd2222abcdqqqqq"; //字符串中"abcd"出现的次数。
```

```
//要求你 自己写一个函数接口，并且写出测试用例。
//完成功能为：求出“abcd”字串出现的次数
//输入：
int getSubCount(char *str, char *substr, int *      mycount)
{
	int ret = 0;
	char *p = str;
	char *sub = substr;
	int count = 0;

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

	//char *p = "abcd11111abcd2222abcdqqqqqabcd";
	//char *p2 = NULL;
	//p2 = p;
	do
	{
		p = strstr(p, sub);
		if (p!= NULL)
		{
			count++;
			//++后缀操作符优先级高，所以先执行*p 操作 然后地址++
			*mycount++;

			p = p + strlen(sub);
		}
		else
		{
			break;
		}
	} while (*p != '\0');
	//printf("count:%d \n", count);

	//mycount 是实参的地址 *（实参的地址）
	*mycount = count;
	return ret;
}
```

05 看图



06 重复的错误何时休

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

void copy_str21_modify(char *from, char *to)
{
	int i = 0;
	if (*from != '\0')
	{
		printf("ddddd");
	}
	for (; *from!='\0'; from++, to++)
	{
		*to = *from;
	}
	*to = '\0';
	printf("to:%s", to);
	printf("from:%s", from);
}
```

```
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);
}

//字符串逆序
int mainaaaa()
{
	char buf1[100] = "abcdefg";
	char to[100];
	copy_str_err(buf1, to);
}


//越界场景
int main000000000000()
{
	char from[5] = "abcde";
	printf("\n %s",from);
	getchar();
	return 0;
}
```

3.7const 专题

1、 const 基础知识（用法、含义、好处、扩展）

```
int main()
{
const int a;    //
int const b;

const char *c;
char * const d; char buf[100]
const char * const    e ;
```

return 0; } Int func1(const) 初级理解：const 是定义常量==》const 意味着只读

含义：
//第一个第二个意思一样 代表一个常整形数
//第三个 c 是一个指向常整形数的指针(所指向的内存数据不能被修改，但是本身可以修改)
//第四个 d 常指针（指针变量不能被修改，但是它所指向内存空间可以被修改）
//第五个 e 一个指向常整形的常指针（指针和它所指向的内存空间，均不能被修改）

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

结论：
//指针变量和它所指向的内存空间变量，是两个不同的概念。。。。。。
//看 const 是放在*的左边还是右边 看 const 是修饰指针变量，还是修饰所指向的内存空变量

3.8 考试强化训练

1、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；70 分

要求 2：编写测试用例。30 分

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

2、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。valude；20 分

功能 2：把奇数位字符挑选出来，组成一个字符串 2，valude 20

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 *souce, char *buf1, char *buf2);

3、键值对（”key = valude”）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 valude；40 分

要求 2：编写测试用例。30 分

要求 3：键值对中间可能有 n 多空格，请去除空格 30 分

注意：键值对字符串格式可能如下：

“key1 = valude1”

“key2 = valude2
“key3 = valude3”
“key4 = valude4”
“key5 = “
“key6 =“
“key7 = “

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

```
int main()
{
getKeyByValude(“key1 = valude1”, ” key1”, buf, &len);
}
```

4 二级指针和多级指针专题

4.1 二级指针的三种内存模型

4.1.1 二级指针输入和输出模型

4.1.2 二级指针三种内存模型

思路：先把二级指针的所用内存模型练一遍，然后我们再探究它的内存模型及本质。

工程开发中二级指针的典型用法

二级指针的第一种内存模型

二级指针的第二种内存模型

二级指针的第三种内存模型

4.1.3 强化两个辅助指针变量挖字符串

思想分享：强化训练到极致

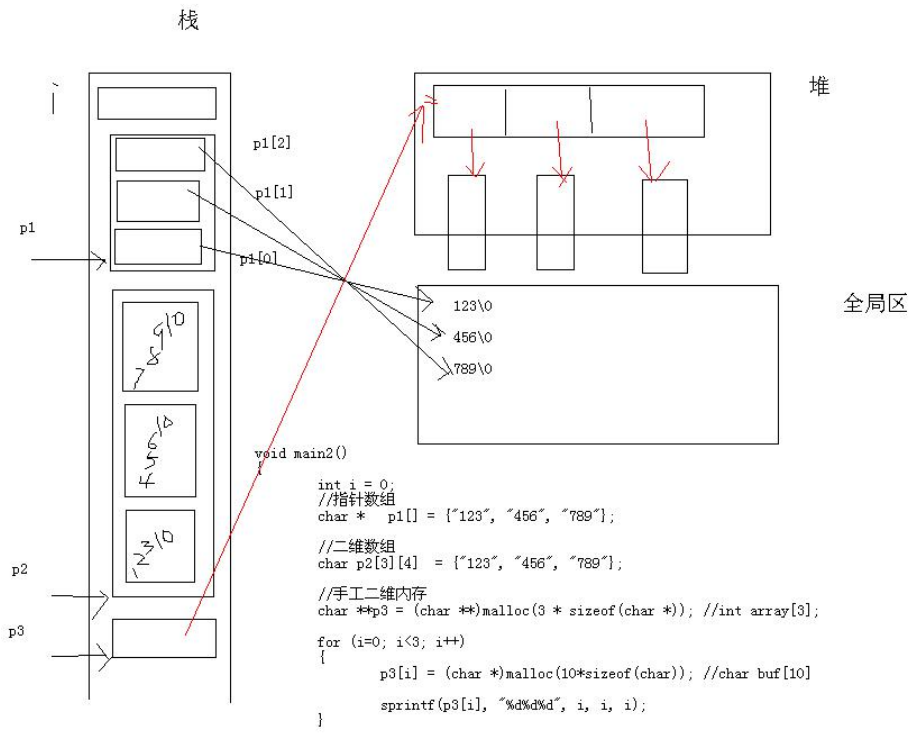
眼高手低 练习到极致 高屋建瓴

一看都会 一练习都错

眼高手低 练习到极致 高屋建瓴

4.1.4 二级指针内存模型建立

```
void main2()
{
	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);
	}
}
```

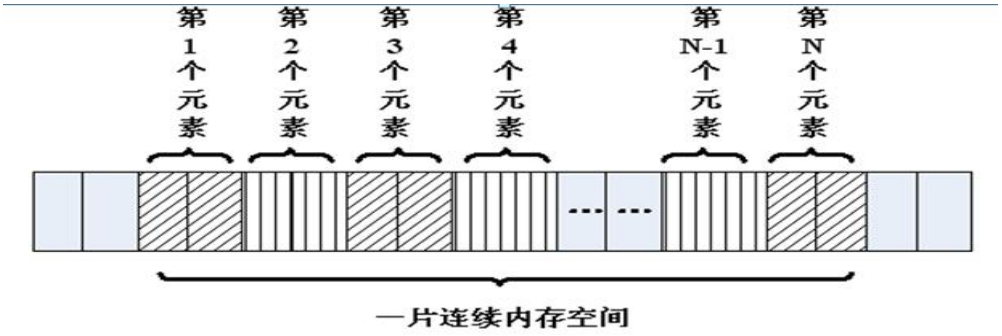


4.2 数组类型和多维数组本质

4.2.1 数组概念

概念

- 1）元素类型角度：数组是相同类型的变量的有序集合 测试指针变量占有内存空间大小
- 2）内存角度：联系的一大片内存空间



数组初始化

- //数组元素的个数可以显示或隐式指定
- //分析数组初始化{0}与 memset 比较

```
int main()
{
	int i = 0;
	int a[10] = {1,2}; //其他初始化为 0
	int b[] = {1, 2};
	int c[20] = {0};

	for (i=0; i<10; i++)
	{
		printf("%d ", a[i]);
	}
	memset(a, 0, sizeof(a));
	getchar();
	return 0;
}
```

数组名的技术盲点

- 1）数组首元素的地址和数组地址是两个不同的概念
- 2）数组名代表数组首元素的地址，它是个常量。
  - 解释如下：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。
- 3）数组首元素的地址和数组的地址值相等
- 4、怎么样得到整个一维数组的地址？

C 语言规定:

```
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？

4.2.2 数组类型、数组指针类型、数组指针类型变量

数组类型

- 1 数据类型分为基础、非基础，思考角度应该发生变化

- 2 C 语言中的数组有自己特定的类型
  - 数组的类型由元素类型和数组大小共同决定
  - 例：int array[5]的类型为 int[5]

```
/*
    typedef int(MYINT5)[5];     //int
    typedef float(MYFLOAT10)[10];
数组定义：
    MYINT5i Array;     int array[5];
MYFLOAT10fArray
*/
```

- 3 定义 数组类型，并用数组类型定义变量

```
int main()
{
    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]);
    }

    getchar();
    return 0;
}
```

数组指针类型

- 数组指针用于指向一个数组

int a[10]

数组名是数组首元素的起始地址，但并不是数组的起始地址

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

//定义数组指针 有两种

1）通过数组类型定义数组指针:

typedef int(ArrayType)[5]; int *a

ArrayType* pointer;

2) 声明一个数组指针类型 typedef int (*MyPointer)[5];

MyPointer myPoint;

3）直接定义：int (*pointer)[n];

pointer 为数组指针变量名

type 为指向的数组的类型

n 为指向的数组的大小

注意这个地方是 type 类型（比如 int （*pointer）[10]）

- 数组指针：用数组类型加*定义一个数组指针

```
//1
{
    int a[5];
    //声明一个数组类型
    typedef int(MYINT5)[5];
    //用数组类型 加*，定义一个数组指针变量
    MYINT5 *array;
    array = &a;
    for (i=0; i<5; i++)
    {
        (*array)[i] = i;
    }
    //
    for (i=0; i<5; i++)
    {
        printf("\n%d %d", a[i], (*array)[i]);
    }
}
```

- 数组指针：定义一个数组指针类型，然后用类型定义变量

```
{
    int b[5];
    //声明一个数组指针类型
    typedef int (*MyPointer)[5];
    //用数组指针类型，去定义一个变量
    MyPointer mypoint ;
```

```
        mypoint= &b;
        for (i=0; i<5; i++)
        {
            (*mypoint)[i] = i;
        }
        //
        for (i=0; i<5; i++)
        {
            printf("\n%d %d", b[i], (*mypoint)[i]);
        }
    }
```

- //3 数组指针：直接定义一个数组指针变量

```
    {
        int c[5];
        //直接声明一个数组指针变量
        int (*pointer)[5] = &c;
        for (i=0; i<5; i++)
        {
            (*pointer)[i] = i;
        }
        for (i=0; i<5; i++)
        {
            printf("\n%d %d", c[i], (*pointer)[i]);
        }
    }
```

4.2.3 多维数组本质技术推演

int a[10];
char myarray[3][5] PK int (*p)[5]
myarray 名称到底是什么？
多维数组 char a[i][j] ==> *(*(a+i)+j)转换技巧分析

```
void main222()
{
    int a[3][5];

    int c[5]; //&c + 1;
    int b[10]; //b 代表数组首元素的地址 &b 代表这个数组的地址 &b+1 相当于 指针后移 4*10 个单位

    //a 代表什么什么那？a 是一个数组指针 指向低维数组的指针
```

```
    //a +1;
    printf("a:%d, a+1:%d \n", a, a+1); //4*5

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

        printf("\n");
        for (i=0; i<3; i++)
        {
            for (j=0; j<5; j++)
            {
                printf("%d \n", a[i][j]);
            }
        }
    }

    //a 的本质是一个数组指针，每次往后跳一维的维数
    {
        int i = 0, j = 0;
        //定义了一个数组指针 变量
        int (*myArrayPoint)[5] ; //告诉编译给我开辟四个字节内存
        myArrayPoint    =    a;
        printf("\n");
        for (i=0; i<3; i++)
        {
            for (j=0; j<5; j++)
            {
                //myArrayPoint[i][j] = ++tmp;
                printf("%d \n", myArrayPoint[i][j]);
            }
        }
    }

    /*
    char cbuf[30]; // cbuf（1 级指针） 代表数组首元素的地址。。。&cbuf（二级指针） 代表整个数组的地址
    char array[10][30]; //array 是二级指针
```

```
    (array+i) //相当于  整个第 i 行的数组地址  //二级指针  &cbuf

    （*(array+i)）//一维数组的首地址  cbuf

    （*(array+i)）+j //相当于第 i 行第 j 列的地址了把。。。。&array[i][j]

    *(（*(array+i)）+j) //相当于第 i 行第 j 列的地址了把。。。。<====>array[i][j]
    */


    system("pause");
}
```

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]**

结论：a 是一个指向 int myarray[5]的数组指针 a+1 向后跳 5*4，跳一行。

4.2.4 多维数组做函数参数退化原因大剖析

```
//证明一下多维数组的线性存储
//线性打印

void printfArray411(int *array, int num)
{
    int i = 0;
    for (i=0; i<num ; i++)
    {
        printf("%d ", array[i]);
```

```
	}
}

void printfArray412(int (*array)[5], int num)
{
	return ;
}

void printfArrr333(int c[3][4][5])
{
	return ;
}
void main()
{
	int a[3][5];
	int c[3][4][5];
	int i , j = 0;
	int tmp = 0;
	for (i=0; i<3; i++)
	{
		for (j=0; j<5; j++)
		{
			a[i][j] = tmp ++;
		}
	}


	printfArray411((int *)a, 15);

	system("pause");
}
```

4.2.5 多维数组做函数参数技术推演

1、 C 语言中只会以机械式的值拷贝的方式传递参数（实参把值传给形参）

int fun(char a[20], size_t b)
{
  printf("%d\t%d",b,sizeof(a));
}

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。
2、二维数组参数同样存在退化的问题
二维数组可以看做是一维数组 二维数组中的每个元素是一维数组 二维数组参数中第一维的参数可以省略 void f(int a[5]) ====》 void f(int a[]); ===》 void f(int* a); void g(int a[3][3])====》 void g(int a[][3]); ====》 void g(int (*a)[3]);
3、等价关系

数组参数	等效的指针参数
一维数组 char a[30]	指针 char*
指针数组 char *a[30]	指针的指针 char **a
二维数组 char a[10][30]	数组的指针 char(*a)[30]

4.3 指针数组的应用场景

指针数组的两种用法（菜单 命令行）
操作系统拉起应用 在框架下干活
字符数组自我结束标志
// NULL 0 '\0'

4.4 强化训练

课堂考试“上黑板”

int sort(char *p[], int count, char **p, int *ncount);
int sort(char *p[], int count, char (*p)[30], int *ncount);
int sort(char (*p)[30], int ncount, char **p, int *ncount);

//把第一种内存模型第二种内存模型结果 copy 到第三种内存模型中，并排序，打印
char ** sort(char **p1, int num1, char (*p)[30], int num2, int *num3);

//

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
```

```
int getArray3_Free(char **p3, int p3num)
{
	int i;
	if (p3 == NULL)
	{
		return -1;
	}
	for (i=0; i<p3num; i++)
	{
		if (p3[i]!=NULL)
		{
			free(p3[i]);
		}
	}
	free(p3);
}


int getArray3_Free2(char ***p3, int p3num)
{
	int i;
	char **tmp = NULL;

	if (p3 == NULL)
	{
		return -1;
	}
	tmp = *p3;

	for (i=0; i<p3num; i++)
	{
		if (tmp[i]!=NULL)
		{
			free(tmp[i]);
		}
	}
	free(tmp);

	*p3 = NULL; //通过间接赋值，去间接的修改实参的值，成 0
}

int getArray3_2(char **myp1, int num1, char    (*myp2)[30], int num2, char ***          myp3,
int *num3)
```

```
{
	int ret = 0;
	int i,j;
	int tmpNum3 = 0;

	char **tmpp3 = NULL;
	char *temp;

	/*
	printf("111111111");
	if (*myp3 ==NULL )
	{
		printf("222222222");
	} */
	printf("33333");
	if (myp1==NULL || myp2==NULL ||num3==NULL || myp3==NULL)
	{
		ret = -1;
		return ret;
	}
	//准备内存
	tmpNum3 = num1 + num2;
	//分配第一维
	tmpp3 = (char **)malloc(tmpNum3 * sizeof(char *));
	if (tmpp3 == NULL)
	{
		return NULL;
	}

	//分配第二维 把第一种内存模型数据和第二种内存模型数据，copy 到第 3 中内存模型
中
	for (i=0; i<num1; i++)
	{
		tmpp3[i] = (char *)malloc(strlen(myp1[i])+1);
		if (tmpp3[i]==NULL)
		{
			puts("out of space");
			return NULL;
		}
		strcpy(tmpp3[i],myp1[i]);
	}
	for (j=0;j<num2;j++,i++)
	{
		tmpp3[i]=(char *)malloc(strlen(myp2[j]) + 1); //note modify
```

```
            if (tmpp3[i]==NULL)
            {
                puts("out of space");
                return NULL;
            }
            strcpy(tmpp3[i],myp2[j]);
        }

        //排序
        for (i=0;i<tmpNum3;i++)
        {
            for (j=i+1;j<tmpNum3;j++)
            {
                if (strcmp(tmpp3[i],tmpp3[j])>0)
                {
                    temp=tmpp3[i];
                    tmpp3[i]=tmpp3[j];
                    tmpp3[j]=temp;
                }
            }
        }

        //通过间接赋值，把结果甩给实参
        *num3=tmpNum3;
        *myp3 = tmpp3; //*0 = 100;
        return ret;
}

char **getArray3(char **myp1, int num1, char    (*myp2)[30], int num2, int *num3)
{
        int i,j;
        int tmpNum3 = 0;

        char **tmpp3 = NULL;
        char *temp;

        if (myp1==NULL || myp2==NULL ||num3==NULL )
        {
            return NULL;
        }
        //准备内存
        tmpNum3 = num1 + num2;
        //分配第一维
        tmpp3 = (char **)malloc(tmpNum3 * sizeof(char *));
```

```
    if (tmpp3 == NULL)
    {
        return NULL;
    }

    //分配第二维 把第一种内存模型数据和第二种内存模型数据，copy 到第 3 中内存模型中
     for (i=0; i<num1; i++)
     {
        tmpp3[i] = (char *)malloc(strlen(myp1[i])+1);
        if (tmpp3[i]==NULL)
        {
            puts("out of space");
            return NULL;
        }
        strcpy(tmpp3[i],myp1[i]);
     }
     for (j=0;j<num2;j++,i++)
     {
         tmpp3[i]=(char *)malloc(strlen(myp2[j]) + 1); //note
         if (tmpp3[i]==NULL)
         {
             puts("out of space");
             return NULL;
         }
         strcpy(tmpp3[i],myp2[j]);
     }

     //排序
     for (i=0;i<tmpNum3;i++)
     {
         for (j=i+1;j<tmpNum3;j++)
         {
             if (strcmp(tmpp3[i],tmpp3[j])>0)
             {
                 temp=tmpp3[i];
                 tmpp3[i]=tmpp3[j];
                 tmpp3[j]=temp;
             }
         }
     }

     *num3=tmpNum3;
    return tmpp3;
```

```
}

void main()
{
	int num3 = 0, i = 0;
	int ret = 0;
	char *p1[] = {"222222", "1111111", "33333333"};
	char p2[4][30] =    {"bbbbb", "aaaaa", "zzzzzz", "ccccccc"};
	char **p3 = NULL;
	char ***myerrp3 = NULL;

	//p3 = getArray3(p1, 3, p2, 4, &num3);
	//ret = getArray3_2(p1,3, p2, 4, &p3, &num3);
	ret = getArray3_2(p1,3, p2, 4, 0, &num3); //错误做法
	if (ret != 0)
	{
		return ;
	}
	for (i=0; i<num3; i++)
	{
		printf("%s \n", p3[i]);
	}

	//getArray3_Free(p3, num3);
	//	p3=NULL;
	getArray3_Free2(&p3, num3);

	printf("p3:%d \n", p3);

	system("pause");
}
```

5 结构体专题

5.1 大纲

01、结构体类型定义及结构体变量定义
char c1,char c2, char name[62]; int age
char name[62]; int age,char c1,char c2
结构体变量的引用 .

结构体变量的指针 ->

02、结构体做函数参数
结构体赋值编译器行为研究
结构体变量做函数参数 PK 结构体指针做函数参数
结构体做函数参数（//结构体赋值和实参形参赋值行为研究）
内存四区调用图画法

//从键盘获取数据，给结构体变量初始化，并排序，打印结构体
stack 上分配结构数组和 heap 上分配结构体数组

03、工程开发中，结构体开发的常见模型及典型错误用法
结构体嵌套一级指针
结构体嵌套二级指针
04、结构体中的深拷贝浅拷贝
问题抛出
解决方法

5.2 结构体类型定义及变量定义

```
/*
//结构体类型定义及结构体变量定义
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
结构体类型定义
*/
//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 main11()
{
	//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, "wangbaoming");

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

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

    printf("hello....\n");
    getchar();
    return 0;
}
```

5.3 结构体做函数参数及结构体数组

```
//测试两个结构体变量之间可以 copy 数据吗？
//t2 = t1;
//测试实参传给形参，编译器的行为
//结果很出人意外
```

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

void printfMyteach01(struct _MyTeacher t)
{
    printf("\nt.name:%s", t.name);
}

void printfMyteach02(struct _MyTeacher *t)
```

```
{
	printf("\nt->name:%s", t->name);
}
```

```
//结构体赋值和实参形参赋值行为研究
int main21()
{
	struct _MyTeacher t1, t2;
	memset(&t1, 0, sizeof(t1));

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

	//测试两个结构体变量之间可以 copy 数据吗？
	//t2 = t1;
	//测试实参传给形参，编译器的行为
	//结果很出人意外
	printfMyteach01(t1);

	printfMyteach02(&t1);

	getchar();
	return 0;
}

//定义结构体数组
int main22()
{
	int i = 0;
	struct _MyTeacher teaArray[3];
	struct _MyTeacher *tmp = NULL;
	for (i=0; i<3; i++)
	{
		strcpy(teaArray[i].name, "aaaaa");
		//printf("%s", teaArray[i].name);
		tmp = &teaArray[i];
		printf("%s", tmp->name);
	}
	getchar();
	return 0;
}
```

例子

从键盘接受数据。。。。并排序

```
int printfArray(struct _MyTeacher *teaArray, int count)
{
	int i = 0;
	//打印
	for (i=0; i<count; i++)
	{
		printf("\n 教师名字： ");
		printf("%s", teaArray[i].name);
		printf("\n 教师年龄： ");
		printf("%d", teaArray[i].age);
	}

}
int main23()
{
	int i = 0, j = 0;
	struct _MyTeacher teaArray[3];

	struct _MyTeacher tmp;

	for (i=0; i<3; i++)
	{
		printf("\n 请键入教师名字： ");
		scanf("%s", teaArray[i].name);
		printf("\n 请键入教师年龄： ");
		scanf("%d", &teaArray[i].age);
	}

	for (i=0; i<3; i++)
	{
		for (j=i+1; j<3; j++)
		{
			if (teaArray[i].age >teaArray[j].age)
			{
				tmp = teaArray[i];
				teaArray[i] = teaArray[j];
				teaArray[j] = tmp;
			}
		}
	}

	//打印
```

```
	for (i=0; i<3; i++)
	{
		printf("\n 教师名字： ");
		printf("%s", teaArray[i].name);
		printf("\n 教师年龄： ");
		printf("%d", teaArray[i].age);
	}

	printf("ddddd\n");
	printfArray(teaArray, 3);

	system("pause");

}
```

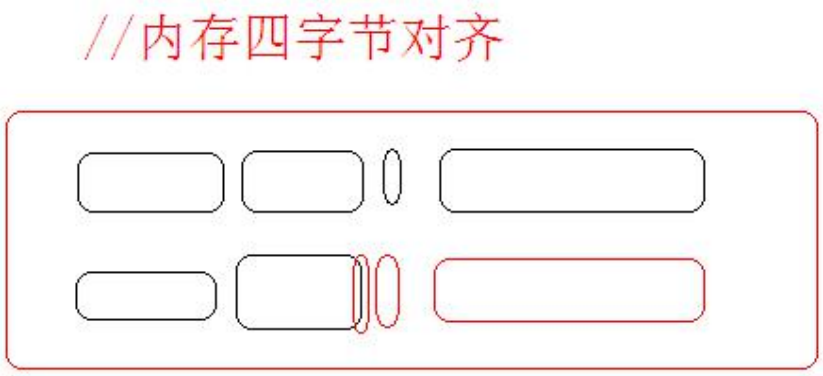
5.4 结构体在工程开发中的应用

```
//测试输入
//测试打印
//测试 malloc
//测试 typdef 用法
//定义结构体数组
```

```
struct _Student
{
	char name[32];
	char tile[32];
};

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

struct _itAdvTeacher
{
	char *name;
	char *tile;
```

```
	int		age;
	char *addr;
	char *p1;
	char **p2;
};


//测试输入
//测试打印
//测试 malloc
//测试 typdef 用法

//定义结构体数组
int main()
{
	int i = 0;
	struct _itTeacher teaArray[3];
	struct _itTeacher *tmp = NULL;
	for (i=0; i<3; i++)
	{
		strcpy(teaArray[i].name, "aaaaa");
		//printf("%s", teaArray[i].name);
		tmp = &teaArray[i];
		printf("%s", tmp->name);
	}
	getchar();
	return 0;
}
```

```
//内存四字节对齐
```



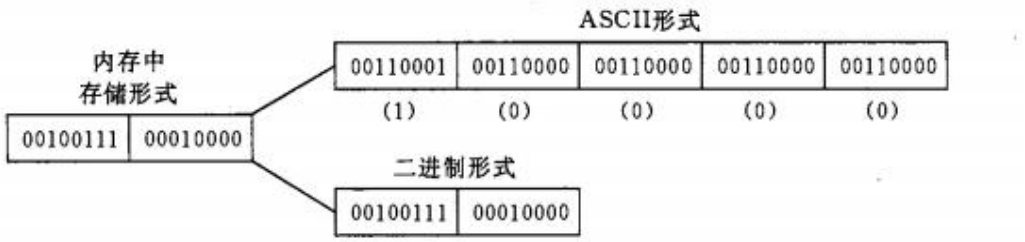
//结构体实参传给形参，也是一个值 copy，相当于 t1 = t2;
//两个结构体变量之间确实是可以 copy，这个是编译器的行为，我们需要顺从

6 文件操作专题

6.1 c 语言文件读写概念

文件分类

- 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

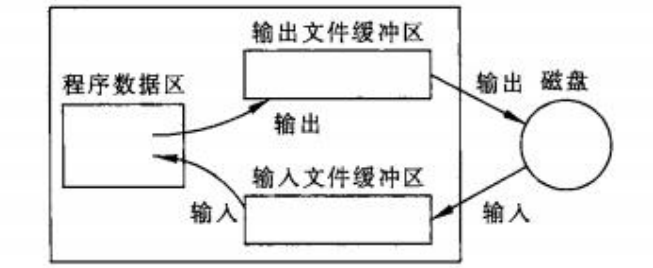


- 每个文件都以文件名为标识，I/O 设备的文件名是系统定义的，如：
- COM1 或 AUX——第一串行口，附加设备

- COM2——第二串行口，此外，还可能有 COM3、COM4 等
- CON——控制台（console），键盘（输入用）或显示器（输出用）
- LPT1 或 PRN——第一并行口或打印机
- LPT2——第二并行口，还可能有 LPT3 等
- NUL——空设备
- 磁盘文件可以由用户自己命名，但上述被系统（windows 和 dos 下均是如此）保留的设备名字不能用作文件名，如不能把一个文件命名为 CON（不带扩展名）或 CON.TXT（不带扩展名）。

- 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
- 缓冲文件系统：高级文件系统，系统自动为正在使用的文件开辟内存缓冲区
- 非缓冲文件系统：低级文件系统，由用户在程序中为每个文件设定缓冲区



-
- 缓冲文件系统理解：文件句柄

```
typedef struct
{short   level;                    /*缓冲区"满"或"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      *curp;         /*指针,当前的指向*/
 unsigned           istemp;        /*临时文件,指示器*/
 short              token;         /*用于有效性检查*/
}FILE;
```

-

6.2 文件操作 API

6.2.1 文件 api 的分类

01）文件读写 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

```
fprintf(fp, "%s = %s\n", pKey, pValue);
```

02）文件控制 api

文件是否结束

文件指针的定位、跳转

```
fseek(fp, 0L, SEEK_END); //把文件指针从 0 位置开始，移动到文件末尾
//获取文件长度;
length = ftell(fp);

fseek(fp, 0L, SEEK_SET);
```

03）api 做项目

6.2.2 标准文件的读写

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
"w"	打开，文件指针指到头，只写
"a"	打开，指向文件尾，在已存在文件中追加

"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，进行追加
"r+"	以读/写方式打开一个已存在的文件
"w+"	以读/写方式建立一个新的文本文件
"a+"	以读/写方式打开一个文件文件进行追加
"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(0 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);
```

DOS 操作系统对同时打开的文件数目是有限制的，缺省值为 5，可以通过修改 CONFIG.SYS 文件改变这个设置。

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");
```

当打开多个文件进行操作，而又要同时关闭时，可采用 fcloseall()函数，它将关闭所有在程序中打开的文件。

int fcloseall();

该函数将关闭所有已打开的文件，将各文件缓冲区未装满的内容写到相应的文件中去，接着释放这些缓冲区，并返回关闭文件的数目。如关闭了 4 个文件，则当执行：

n=fcloseall(); 时，n 应为 4。

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 fgetchar(void);
int fputc(int ch,FILE *stream);
int fputchar(int ch);
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"
main()
{
    FILE *fp;
    ch ch;
    if((fp=fopen("myfile.tex","r"))==NULL)
    {
        printf("file cannot be opened\n");
        exit(1);
    }
        while((ch=fgetc(fp))!=EOF) fputc(ch,stdout);
        fclose(fp);
}
```

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。又比如：

putc(ch,fp);

该函数执行结构，将把 ch 表示的字符送到流指针 fp 指向的文件中去。

在 TC 中，putc()等价于 fput(),getc()等价于 fgetc()。

putchar(c)相当于 fputc(c,stdout)；getchar()相当于 fgetc(stdin)。

注意，这里使用 char ch,其实是不科学的，因为最后判断结束标志时，是看 ch!=EOF,而 EOF 的值为-1，这显然和 char 是不能比较的。所以，某些使用，我们都定义成 int ch。

(2).读写文件中字符串的函数

```
char *fgets(char *string,int n,FILE *stream);
char *gets(char *s);
int fprintf(FILE *stream,char *format,variable-list);
int fputs(char *string,FILE *stream);
int fscanf(FILE *stream,char *format,variable-list);
```

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"
main()
{
    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);
}
```

gets()函数执行时，只要未遇到换行符或文件结束标志，将一直读下去。因此读到什么时候为止，需要用户进行控制，否则可能造成存储区的溢出。

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()和 fscanf()同 printf()和 scanf()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；fscanf()是从文件输入。

下面程序是向文件 test.dat 里输入一些字符：

```
#include<stdio.h>
main()
 {
       char *s="That's good news";
       int i=617;
       FILE *fp;
       fp=fopne("test.dat", "w");
       fputs("Your score of TOEFLis",fp);
       fputc(':', fp);
       fprintf(fp, "%d\n", i);
       fprintf(fp, "%s", s);
       fclose(fp);
 }
```

用 DOS 的 TYPE 命令显示 TEST.DAT 的内容如下所示:

屏幕显示

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

下面的程序是把上面的文件 test.dat 里的内容在屏幕上显示出来：

```
  #include<stdio.h>
  main()
  {
        char *s, m[20];
        int i;
        FILE       *fp;
        fp=fopen("test.dat", "r");
        fgets(s, 24, fp);
        printf("%s", s);
        fscanf(fp, "%d", &i);
        printf("%d", i);
        putchar(fgetc(fp));
        fgets(m, 17, fp);
        puts(m);
        fclose(fp);
        getch();
  }
```

运行后屏幕显示:

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

4.清除和设置文件缓冲区

(1).清除文件缓冲区函数：

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

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。

flushall()将清除所有打开文件所对应的文件缓冲区。

(2).设置文件缓冲区函数

```
void setbuf(FILE *stream,char *buf);
void setvbuf(FILE *stream,char *buf,int type,unsigned size);
```

这两个函数将使得打开文件后，用户可建立自己的文件缓冲区，而不使用 fopen()函数打开文件设定的默认缓冲区。

对于 setbuf()函数，buf 指出的缓冲区长度由头文件 stdio.h 中定义的宏 BUFSIZE 的值决定，缺省值为 512 字节。当选定 buf 为空时，setbuf 函数将使的文件 I/O 不带缓冲。而对 setvbuf 函数，则由 malloc 函数来分配缓冲区。参数 size 指明了缓冲区的长度(必须大于 0),而参数 type 则表示了缓冲的类型，其值可以取如下值：

type 值	含义
_IOFBF	文件全部缓冲，即缓冲区装满后，才能对文件读写
_IOLBF	文件行缓冲，即缓冲区接收到一个换行符时，才能对文件读写
_IONBF	文件不缓冲，此时忽略 buf,size 的值，直接读写文件，不再经过文件缓冲区缓冲

5.文件的随机读写函数

前面介绍的文件的字符/字符串读写，均是进行文件的顺序读写，即总是从文件的开头开始进行读写。这显然不能满足我们的要求，C 语言提供了移动文件指针和随机读写的函数，它们是：

(1).移动文件指针函数：

```
long ftell(FILE *stream);
int rewind(FILE *stream);
fseek(FILE *stream,long offset,int origin);
```

函数 ftell()用来得到文件指针离文件开头的偏移量。当返回值是-1 时表示出错。

rewind()函数用于文件指针移到文件的开头，当移动成功时，返回 0，否则返回一个非 0 值。

fseek()函数用于把文件指针以 origin 为起点移动 offset 个字节，其中 origin 指出的位置可有以下几种：

origin	数值	代表的具体位置
SEEK_SET	0	文件开头
SEEK_CUR	1	文件指针当前位置
SEEK_END	2	文件尾

例如：

fseek(fp,10L,0);

把文件指针从文件开头移到第 10 字节处，由于 offset 参数要求是长整型数，故其数后带 L。

fseek(fp,-15L,2);

把文件指针从文件尾向前移动 15 字节。

(2).文件随机读写函数

int fread(void *ptr,int size,int nitems,FILE *stream);

int fwrite(void *ptr,int size,int nitems,FILE *stream);

fread()函数从流指针指定的文件中读取 nitems 个数据项，每个数据项的长度为 size 个字节，读取的 nitems 数据项存入由 ptr 指针指向的内存缓冲区中，
在执行 fread()函数时，文件指针随着读取的字节数而向后移动，最后移动结束的位置等于实际读出的字节数。该函数执行结束后，将返回实际读出的数据项数，
这个数据项数不一定等于设置的 nitems，因为若文件中没有足够的数据项，或读中间出错，都会导致返回的数据项数少于设置的 nitems。当返回数不等于 nitems 时
，可以用 feof()或 ferror()函数进行检查。

fwrite()函数从 ptr 指向的缓冲区中取出长度为 size 字节的 nitems 个数据项，写入到流指针 stream 指向的文件中，执行该操作后，文件指针将向后移动，
移动的字节数等于写入文件的字节数目。该函数操作完成后，也将返回写入的数据项数。

6.2.3 非标准文件的读写

这类函数最早用于 UNIX 操作系统,ANSI 标准未定义,但有时也经常用到,DOS 3.0 以上版本支持这些函数。它们的头文件为 io.h。

由于我们不常用这些函数，所以在这里就简单说一下。

1.文件的打开和关闭

open()函数的作用是打开文件,其调用格式为:

int open(char *filename, int access);

该函数表示按 access 的要求打开名为 filename 的文件,返回值为文件描述字,其中 access 有两部分内容:

基本模式和修饰符，两者用" "("或")方式连接。修饰符可以有多个，但基本模式只能有一个。

access 的规定

基本模式	含义	修饰符	含 义
O_RDONLY	只读	O_APPEND	文件指针指向末尾

O_WRONLY	只写	O_CREAT	文件不存在时创建文件，属性按基本模式属性
O_RDWR	读写	O_TRUNC	若文件存在，将其长度缩为 0，属性不变
O_BINARY	打开一个二进制文件		
O_TEXT	打开一个文字文件		

---

open()函数打开成功，返回值就是文件描述字的值(非负值)，否则返回-1。
close()函数的作用是关闭由 open()函数打开的文件，其调用格式为:

```
int close(int handle);
```

该函数关闭文件描述字 handle 相连的文件。

2.读写函数

```
int read(int handle, void *buf, int count);
```

read()函数从 handle(文件描述字)相连的文件中，读取 count 个字节放到 buf 所指的缓冲区中，

返回值为实际所读字节数，返回-1 表示出错。返回 0 表示文件结束。

write()函数的调用格式为:

```
int write(int handle, void *buf, int count);
```

write()函数把 count 个字节从 buf 指向的缓冲区写入与 handle 相连的文件中，返回值为实际写入的字节数。

3.随机定位函数

lseek()函数的调用格式为:

```
int lseek(int handle, long offset, int fromwhere);
```

该函数对与 handle 相连的文件位置指针进行定位,功能和用法与 fseek()函数相同。

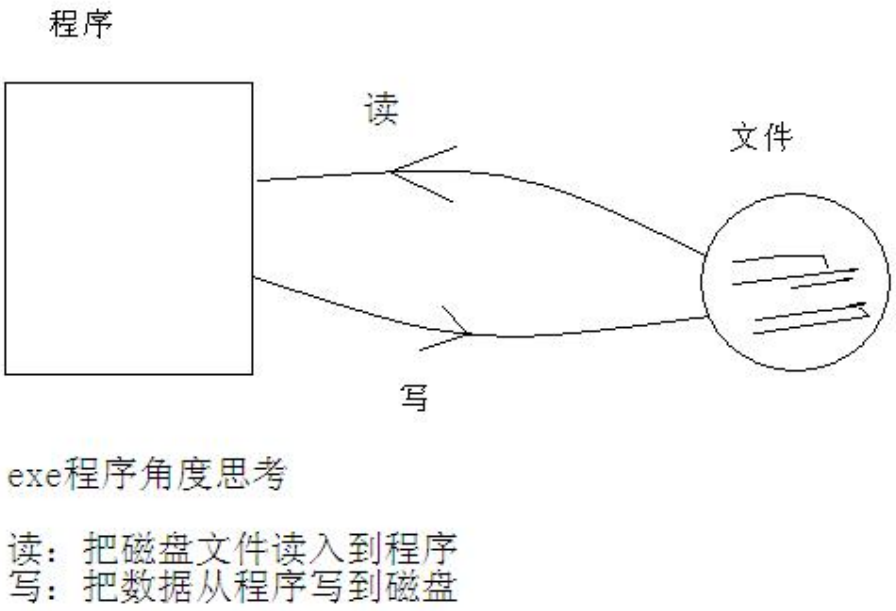
tell()函数的调用格式为:

```
long tell(int handle);
```

该函数返回与 handle 相连的文件现生位置指针，功能和用法与 ftell()相同

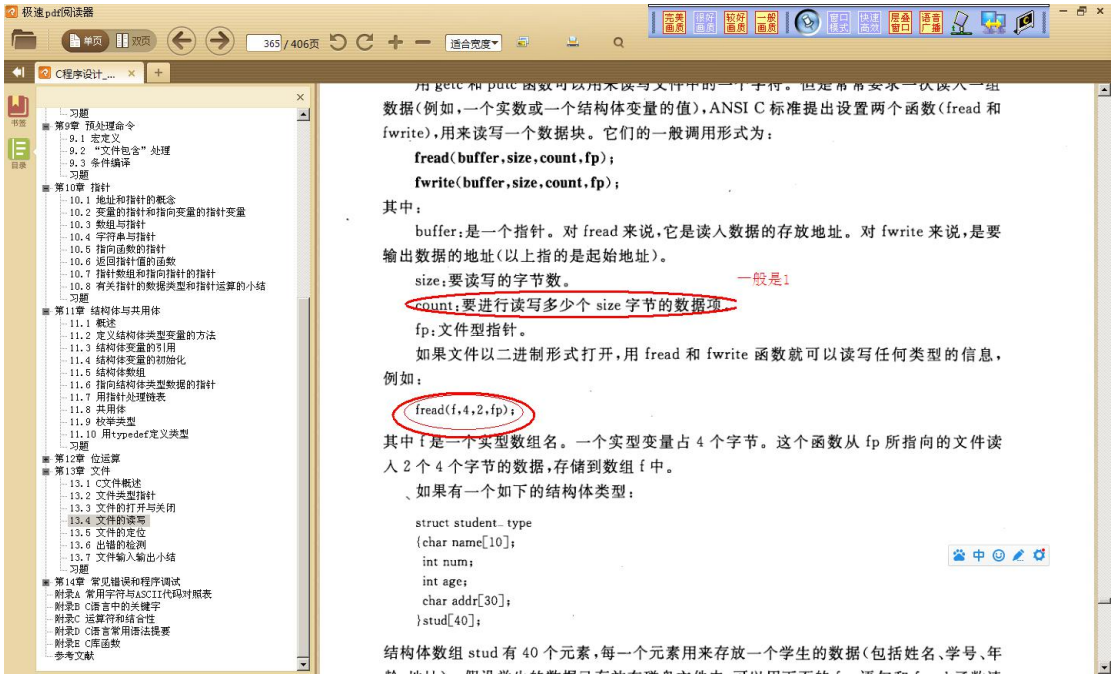
6.2.4 注意点

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



项目开发中参考 fgets 函数的实现方法

fgets(buf, bufMaxLen, fp);
对fgets函数来说，n必须是个正整数，表示从文件按中读出的字符数不超过n-1，存储到字符数组str中，并在末尾加上结束标志’\0’，换言之，n代表了字符数组的长度，即sizeof(str)。如果读取过程中遇到换行符或文件结束标志，读取操作结束。若正常读取，返回指向str代表字符串的指针，否则，返回NULL（空指针）。



6.3 文件操作案例-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。

6.4 文件操作案例-大文件加解密

功能实现分析

1、数据加密解密接口测试

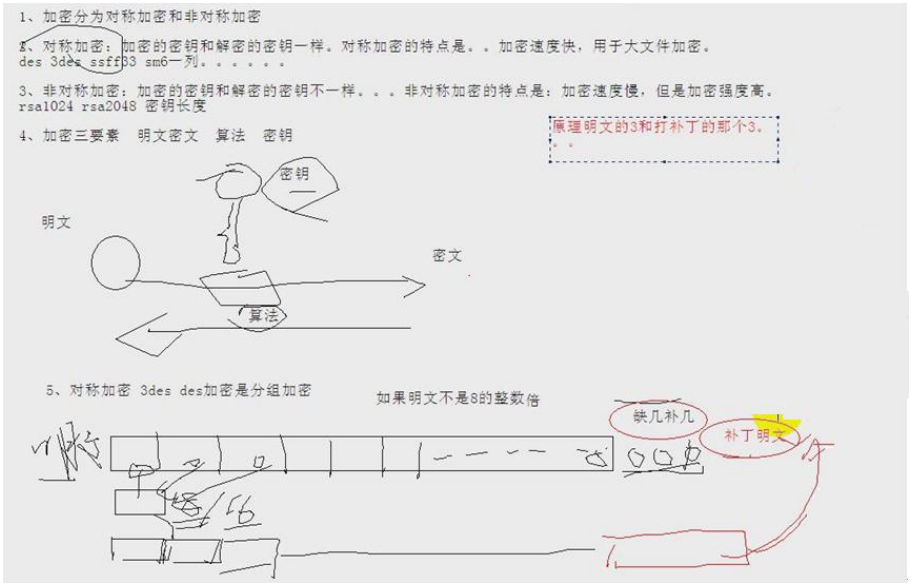
2、数据加密过程分析

文件数据的 movecopy + 数据加密

3、数据加解密功能集成

数据加密和解密分为两个版本 打 padding 和不打 padding

数据加密解密原理



7 C 接口的封装和设计专题

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：

```
c====》应用
win/linux 系统编程
api
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。

DLL 分配的内存如何在 EXE 里面释放

总结下面几个要点：

1. 保证内存分配和清除的统一性：如果一个 DLL 提供一个能够分配内存的函数，那么这个 DLL 同时应该提供一个函数释放这些内存。数据的创建和清除应该在同一个层次上。

曾经遇到过这样的例子：在 dll 中分配了一块内存,通过 PostMessage 将其地址传给应用。然后应用去释放它,结果总是报异常。

2.如果 exe 用 MFC Appwizard 方式生成， dll 用 win32 方式生成，则运行时会出现错误。进一步用单步跟踪，发现 mfc 方式和 win32 方式下的 new 操作符是用不同方式实现的，源程序分别在 VC 目录的文件 Afxmem.cpp 和 new.cpp 中。有兴趣的话可以自己跟踪一下。

因为 dll 输出函数后，并不知道是哪一个模拟调用它，因此 new 和 delete 配对时最好在一个文件中，这样可以保证一致性。

3. 问题主要在于 DLL 和 EXE 主程序中分配内存的堆不一样，你可以不用 new 和 delete，而是用

1） ::HeapAlloc(::GetProcessHeap(),...)和::HeapFree(::GetProcessHeap(),...)

2） ::GlobalAlloc()和::GlobalFree()

这两对 API，这样无论在 DLL 中还是在主程序中都是在进程默认堆中分配，就不会出错了。

4. 还有一个办法，就是把 dll 的 Settings 的 C/C++选项卡的 Code Generation 的 Use Run-time liberary 改成 Debug Multithreaded DLL，在 Release 版本中改成 Multithreaded DLL，就可以直接使用 new 和 delete 了。不过 MFC 就不能用 Shared 模式了。