Luffy Li (李壮)

嵌入式学习笔记（二）

C语言高级专题

基于S5PV210

[一、内存这个大话题 1](#_Toc26340388)

[1.1程序运行为什么需要内存？ 1](#_Toc26340389)

[1.1.1.计算机程序运行的目的 1](#_Toc26340390)

[1.1.2.计算机程序的运行过程 1](#_Toc26340391)

[1.1.3.哈佛结构和冯诺依曼结构 1](#_Toc26340392)

[1.1.4.程序运行为什么需要内存？ 1](#_Toc26340393)

[1.1.5.内存管理 1](#_Toc26340394)

[1.2内存位宽 2](#_Toc26340395)

[1.2.1.内存的逻辑抽象图 2](#_Toc26340396)

[1.2.2.内存位宽 2](#_Toc26340397)

[1.3 内存编址和寻址、内存对齐 2](#_Toc26340398)

[1.3.1.内存编址 2](#_Toc26340399)

[1.3.2.关键：内存编址是以字节为单位的 2](#_Toc26340400)

[1.3.3.内存和数据类型的关系 2](#_Toc26340401)

[1.3.4.内存对齐 3](#_Toc26340402)

[1.4 C语言如何操作内存 3](#_Toc26340403)

[1.4.1. C语言对内存地址的封装 3](#_Toc26340404)

[1.4.2. C语言数据类型的本质含义：表示内存格子的个数（每个格子1个字节）和解析方法。 3](#_Toc26340405)

[1.4.3.用指针来间接访问内存 3](#_Toc26340406)

[1.4.4.用数组来管理内存 3](#_Toc26340407)

[1.5 内存管理之结构体 4](#_Toc26340408)

[1.5.1.最简单的数据结构：数组 4](#_Toc26340409)

[1.5.2.结构体隆重登场： 4](#_Toc26340410)

[1.5.3.题外话：结构体内嵌指针实现面向对象 4](#_Toc26340411)

[1.6 内存管理之栈 4](#_Toc26340412)

[1.6.1.什么是栈（Stack） 4](#_Toc26340413)

[1.6.2.栈管理内存的特点（小内存、自动化） 4](#_Toc26340414)

[1.6.3.栈的应用举例：局部变量 4](#_Toc26340415)

[1.7内存管理之堆 5](#_Toc26340416)

[1.7.1.什么是堆（heap） 5](#_Toc26340417)

[1.7.2.堆管理内存的特点（大块内存、手工分配&使用&释放） 5](#_Toc26340418)

[1.7.3.C语言操作堆内存的接口（malloc、free） 5](#_Toc26340419)

[1.7.4.堆的优势和劣势（管理大块内存、灵活、容易内存泄露） 6](#_Toc26340420)

[1.8复杂数据结构 6](#_Toc26340421)

[1.8.1.链表、哈希表（散列表）、二叉树、图等 6](#_Toc26340422)

[1.8.2.为什么需要复杂数据结构 6](#_Toc26340423)

[1.8.3.数据结构和算法（可以用程序实现的代码）的关系 6](#_Toc26340424)

[二、C语言位操作 6](#_Toc26340425)

[2.1位操作符 7](#_Toc26340426)

[2.1.1.位与& 7](#_Toc26340427)

[2.1.2.位或| 7](#_Toc26340428)

[2.1.3.位取反~ 7](#_Toc26340429)

[2.1.4.位异或^ 7](#_Toc26340430)

[2.1.5.左移位<< 与右移位>> 7](#_Toc26340431)

[2.2位与位或位异或在操作寄存器时的特殊作用 8](#_Toc26340432)

[2.2.1寄存器操作的要求（特定位改变而不影响其他位） 8](#_Toc26340433)

[2.2.2特定位清零用& 8](#_Toc26340434)

[2.2.3特定位置1用| 8](#_Toc26340435)

[2.2.4特定位取反用^ 8](#_Toc26340436)

[2.3如何用位运算构建特定二进制数 8](#_Toc26340437)

[2.3.1.寄存器位操作经常需要特定位给特定值 8](#_Toc26340438)

[2.3.2.使用移位获取特定位为1的二进制数 9](#_Toc26340439)

[2.3.3.再结合位取反获取特定位为0的二进制数 9](#_Toc26340440)

[2.3.4.总结：位与、位或结合特定二进制数即可完成寄存器位操作需求 9](#_Toc26340441)

[2.4位操作实战 9](#_Toc26340442)

[三、指针才是C的精髓 11](#_Toc26340443)

[3.1指针到底是什么 11](#_Toc26340444)

[3.1.1.指针变量和普通变量的区别 11](#_Toc26340445)

[3.1.2.为什么需要指针 11](#_Toc26340446)

[3.1.3.指针使用三部曲：定义指针变量、关联指针变量、解引用 11](#_Toc26340447)

[3.2指针带来的一些符号的理解 11](#_Toc26340448)

[3.2.1.星号\* 11](#_Toc26340449)

[3.2.2.取地址符& 11](#_Toc26340450)

[3.2.3.指针定义并未初始化、与指针定义然后赋值的区别 12](#_Toc26340451)

[3.2.4.左值与右值 12](#_Toc26340452)

[3.3野指针问题 12](#_Toc26340453)

[3.3.1.神马是野指针？哪里来的？有什么危害？ 12](#_Toc26340454)

[3.3.2.怎么避免野指针？ 12](#_Toc26340455)

[3.3.3.NULL到底是什么？ 13](#_Toc26340456)

# 一、内存这个大话题

## 1.1程序运行为什么需要内存？

### 1.1.1.计算机程序运行的目的

程序 = 代码 + 数据

代码就是函数，表示加工数据的动作。

数据包括全局变量和局部变量，表示被加工的东西。

程序运行的目的要么重在数据结果（有返回值），要么重在过程（无返回值），要么既重视结果又重视过程。

### 1.1.2.计算机程序的运行过程

计算机程序的运行过程，其实就是程序中很多个函数相继运行的过程。程序是由很多个函数组成的，程序的本质就是函数，函数的本质就是加工数据的动作。

### 1.1.3.哈佛结构和冯诺依曼结构

哈佛结构：哈佛结构就是将程序的代码和数据分开存放的一种结构，而他们存放的位置可以是相同的也可以是不同的（ROM&RAM或者RAM），总之只要是分成两个部分单独访问的结构都可以叫哈佛结构。（例如：51的程序运行时，代码放在ROM（NorFlash）中原地运行，而数据则存放在RAM中随代码动作而变动；而S5PV210程序运行时，代码和数据都在DRAM中运行，但是DRAM中又划分了代码段和数据段，二者互不干扰。）哈佛结构的特点就是代码和数据单独存放，使之不会互相干扰，进而当程序出BUG时，最多只会修改数据的值（因为代码部分是只读的，不可改写），而不会修改程序的执行顺序。因此，这种结构大量应用在嵌入式编程。

冯诺依曼结构：冯诺依曼结构是将代码和数据统一都放在RAM中，他们之间一般是按照程序的执行顺序依次存储。这样就会导致一个问题，如果程序出BUG，由于程序没有对代码段的读写限定，因此，它将拥有和数据一样的读写操作权限。于是就会很容易的死机，一旦代码执行出现一点改变就会出现非常严重的错误。但是，冯诺依曼结构的好处是可以充分利用有限的内存空间，并使CPU对程序的执行十分的方便，不用来回跑。

### 1.1.4.程序运行为什么需要内存？

对于S5PV210的程序来说，程序运行时要存放代码和数据，代码放在DRAM的只读权限代码段，数据放在DRAM的可读可写数据段，程序要跑，内存是必要条件 。

### 1.1.5.内存管理

从OS角度讲：OS掌握所有的硬件内存，因为内存很大，所以OS把内存分成1个1个的页面（其实就是分块，一般是4KB），然后以页面为单位来管理。页面内用更细小的方式来以字节为单位管理。（OS的内存管理原理复杂，对于我们使用OS的人来说，我们无需了解细节。OS为我们提供了内存管理的一些接口，我们只需要用相应的API即可管理内存。例如C语言中使用malloc、free这些接口来管理内存）。

在没有OS时，也就是裸机程序中程序需要直接操作内存，编程者需要自己计算内存的使用和安排。

从语言角度讲：不同语言提供了不同的操作内存的接口

汇编：根本没有内存管理，汇编中操作内存时直接使用内存地址（如0xd0020010）

C：C语言编译器帮我们管理内存地址，我们都是通过编译器通过的变量名来访问内存的，OS下如果需要大块内存，可以通过API（mallos、free）来访问内存。裸机程序中需要大块内存需要自己定义数组等来解决。

C++：C++对内存的使用进一步封装。我们可以用new来创建对象（其实就是为对象分配内存），然后使用完了用delete来删除对象（其实就是释放内存）。所以C++比C更容易一些。但是C++中的内存管理还是要靠程序员自己来做，例如需要使用delete删除对象释放内存，如果忘记，就会造成内存不能释放，就是所谓的内存泄露。

JAVA/C#等：这些语言不直接操作内存，而是通过虚拟机来操作内存。这样虚拟机作为我们程序员的代理，来帮我们处理内存的释放工作。如果程序申请了内存，使用后忘记释放，那么虚拟机会帮我们释放。看起来，JAVA/C#比C/C++有优势，但是虚拟机回收内存的机制也是要付出一定的代价。

## 1.2内存位宽

### 1.2.1.内存的逻辑抽象图

提到内存，脑中要有一张逻辑图。这张图是一行行大小相等的格子，对于32位内存来说，一行就是4个字节。CPU要访问一个int型数据，则首先取地址,这里的地址指的是int型数据单元的首地址，即4字节中的首字节的地址，然后就可以读取到这4个字节空间中所保存的数据。

### 1.2.2.内存位宽

从硬件角度：硬件的内存实现本身就是有宽度的，也就是内存条本身就有8位、16位等。需要注意的是，内存芯片之间可以并联，通过并联后8位内存芯片可以做出来16位、32位的硬件内存。

从逻辑角度：内存位宽在逻辑上是任意的，甚至逻辑上内存的位宽可以是24位，但没必要。从逻辑角度，不管内存位宽多少，直接操作即可。但因为所有的逻辑操作都是要硬件实现，所以还是要尊重硬件内存位宽。

## 1.3 内存编址和寻址、内存对齐

### 1.3.1.内存编址

在程序运行中，CPU实际只认识内存地址，而不关心这个地址所代表的空间在哪里、怎么分布等这些实体问题，因为硬件设计保证了这个地址就一定能找到这个格子，所以内存单元的2个概念：地址和空间是内存单元的两个方面。

### 1.3.2.关键：内存编址是以字节为单位的

### 1.3.3.内存和数据类型的关系

数据类型是用来定义变量的，而这些变量需要在内存中存储和运算。所以数据类型必须和内存相匹配才能获得最好的性能。

在32位系统中，定义变量最好使用32位的int，因为这样效率高。原因是32的数据类型配合32位的内存是可以实现32位CPU最好的性能。当定义8位的char时，CPU访问内存的效率其实是不高的。在很多情况下，我们定义的8位char变量，编译器会帮我们分配32位内存来存储这个char变量，也就是说浪费了24位的内存，但是效率高。

### 1.3.4.内存对齐

内存的对齐访问不是逻辑问题，是硬件问题。从硬件角度来说，32位的内存它0、1、2、3四个单元本身逻辑上就有相关性，这4个字节组合起来当做一个int，硬件上就是合适的，效率就高。

## 1.4 C语言如何操作内存

### 1.4.1. C语言对内存地址的封装

变量名即对内存地址的封装。指针即保存这个地址的变量。函数名实质就是一段代码的首地址。

### 1.4.2. C语言数据类型的本质含义：表示内存格子的个数（每个格子1个字节）和解析方法。

(1)决定内存格子的个数：如果给一个地址0x30000000，那么这个地址即一个格子。如果int定义它，这个地址就会扩展为4个格子。

(2)解析方法：（int）0x30000000含义就是从0x30000000开始的4个格子连起来共同存放的一个int型数据。（float）0x30000000含义就是从0x30000000开始的4个格子连起来共同存放的一个float型数据。

### 1.4.3.用指针来间接访问内存

关于数据类型（不管是普通变量类型int、float等，还是指针变量类型int \*、float \* 等），只要记住：

类型只是对后边数字或者符号（代表的都是内存地址）所表征的内存的一种长度规定和解析方法规定而已。

C语言中的指针，全名叫指针变量，指针变量其实很普通变量没有任何区别。譬如int a和int \*p其实没有任何区别，a和p都代表一个内存地址（譬如是0x20000000），但是这个内存地址（0x20000000）的长度和解析方法不同。a是int型所以a的长度是4字节，解析方法是按照int的规定来的(以0x20000000开头的连续的4个字节的空间中存了一个int型的数)；p是int \*类型，所以长度是4字节，解析方法是int \*的规定来的（以0x20000000开头的连续4字节的空间中存储了1个地址，这个地址所代表的内存单元中存放的是一个int类型的数）。在32位系统中，指针变量永远占4个字节的内存空间。

### 1.4.4.用数组来管理内存

数组管理内存和变量其实没有本质区别，只是符号的解析方法不用。（普通变量、数组、指针变量其实都没有本质差别，都是对内存地址的解析，只是解析方法不一样）。

int a; //编译器分配4个字节长度给a，并且把首地址和符号a绑定起来。

int b[10]; //编译器分配40个字节长度给b，并且把首元素的首地址和符号b绑定起来。

数组中第一个元素（b[0]）就称为首元素；每一个元素都是类型都是int，所以长度都是4个字节，其中第一个字节的地址就称为首地址；首元素b[0]的首地址就称为首元素首地址。

## 1.5 内存管理之结构体

数据结构：是一门研究数据在内存中如何分布的学问。

### 1.5.1.最简单的数据结构：数组

数组的特点：类型相同、意义相关

数组的优势：数组比较简单，访问使用下标，可以随机访问（就是可以通过下标随机访问需要访问的元素）。

数组的缺陷：（1）数组中元素类型必须相同 （2）数组大小必须在定义时给出，而且一旦给出不能更改。

### 1.5.2.结构体隆重登场：

结构体发明出来就是为了解决数组的第（1）个缺陷。

结构体和数组的本质差异还是在于怎么找变量地址的问题。

### 1.5**.3.题外话：结构体内嵌指针实现面向对象**

C语言作为面向过程的语言，可以通过结构体内嵌指针实现面向对象的代码。

当然，面向对象的语言更为简单直观。

struct s

{

int age // 普通变量

void (\*pFunc)(void); // 函数指针，指向 void func(void)这类的函数

};

使用这样的结构体就可以实现面向对象。

这样包含了函数指针的结构体就类似于面向对象中的class，结构体中的变量类似于class中的成员变量，结构体中的函数指针类似于class中的成员方法。

## 1.6 内存管理之栈

### 1.6.1.什么是栈（Stack）

栈是一种数据结构，C语言中使用栈来存放局部变量。

### 1.6.2.栈管理内存的特点（小内存、自动化）

先进后出 FILO（First In Last Out） 栈

先进先出 FIFO（First In First Out） 队列

栈的特点是入口即出口，只有一个口，另一个口是堵死的。所以先进去的必须后出来

队列的特点是入口和出口都有，必须从入口进，从出口出，所以先进去的必须先出来，否则就堵住后边的。

### 1.6.3.栈的应用举例：局部变量

C语言中的局部变量是用栈来实现的。

我们在C语言中定义一个局部变量时（int a），编译器会在栈中分配一段空间（4字节）给这个局部变量用（分配时栈顶指针会移动给出空间，给局部变量a用的意思就是，将这字节的栈内存的内存地址和我们定义的局部变量名a给关联起来），对应栈的操作是入栈。

注意：这里栈指针的移动和内存分配都是自动的。

然后等我们函数退出时，局部变量就会灭亡。对应栈的操作就是弹栈（出栈）。出栈时也是栈顶指针移动将栈空间中与a关联的那4个字节空间释放。这个动作也是自动的，不需要写代码干预。

栈的优点：入栈和出栈都由C语言自动完成。

分析一个细节：C语言中，定义局部变量时如果未初始化，则值是随机的，为什么？

定义局部变量，其实就是在栈中通过移动栈指针来给程序提供一个内存空间和这个局部变量名绑定。因为这段内存空间在栈上，而栈内存是反复使用的（脏的，上次用完没清零的），所以说使用栈来实现的局部变量定义时如果不显式初始化，值就是脏的（也就是随机值）。

栈的约束：预定栈大小不灵活，怕溢出

首先，栈是有大小的。所以栈内存大小不好设置。如果太小怕溢出，太大怕浪费内存。（这个缺点有点像数组）

其次，栈的溢出危害很大，一定要避免。所以我们在C语言中定义局部变量时不能定义太多或者太大（譬如不能定义局部变量时 int a[10000]; 使用递归来解决问题时一定要注意递归收敛）

## 1.7内存管理之堆

### **1.7.1.**什么是堆（heap）

内存管理对OS来说是一件非常复杂的事，因为首先内存容量大，其次内存需求在时间和大小块上没有规律（OS上运行着几十、几百、几千个进程随时都会申请或者释放内存，申请或者释放的内存块大小随意）。

堆这种内存管理方式特点就是自由（随时申请、释放；大小块随意）。堆内存是OS划归给堆管理器（OS中的一段代码，属于OS的内存管理单元）来管理的，然后向使用者（用户进程）提供API（malloc、free）来使用堆内存。

我们会在需要内存容量比较大，需要反复使用及释放时，会使用堆内存。很多数据结构（譬如链表）的实现都需要使用堆内存。

### **1.7.**2.堆管理内存的特点（大块内存、手工分配&使用&释放）

特点1：容量不限（常规使用的需求容量都能满足）。

特点2：申请及释放都需要手工进行，手工进行的含义就是需要写代码明确申请malloc和释放free。如果申请内存并使用后未释放，这段内存就丢失了（在堆管理器的记录中，这段内存仍然属于你这个进程，但是进程自己又以为这段内存已经不用了，再用的时候又会去申请新的内存块，这就叫吃内存。），称为内存泄露。。。在C/C++语言中，内存泄露是最严重的程序bug，这也是Java/C#等语言比C/C++优秀的地方。

### **1.7.**3.C语言操作堆内存的接口（malloc、free）

堆内存释放时最简单，直接调用free释放即可。 void free(void \*ptr)

堆内存申请时，有3个可选择的类似功能的函数：malloc、calloc、realloc

void \*malloc(size\_t size);

void \*calloc(size\_t nmemb, size\_t size); //nmemb个单元，每个单元size字节

void \*realloc(void \*ptr, size\_t size); //改变原来申请的空间的大小

譬如要申请10个int元素的内存：

malloc(40); malloc(10\*sizeof(int));

calloc(10, 4); calloc(10, sizeof(int));

数组定义时必须同时给出数组元素的个数（数组大小），而且一旦定义再无法更改。在Java等高级语言中，有一些语法技巧可以更改数组大小，但其实这只是一种障眼法。它的工作原理是：先重新创建一个新的数组大小为要更改后的数组，然后将原数组的所有元素复制进新的数组，然后释放掉原数组，最后返回新的数组给用户。

堆内存申请时必须给定大小，然后一旦申请完成大小不能更改，如果要变更，只能通过realloc接口。realloc的实现原理类似于上边说的Java中的可变大小的数组的方式。

### **1.7.**4.堆的优势和劣势（管理大块内存、灵活、容易内存泄露）

优势：灵活

劣势：需要人为处理各种细节，所以容易出错

## 1.8复杂数据结构

### 1.8.1.链表、哈希表（散列表）、二叉树、图等

链表是最重要的，链表在Linux内核中使用非常多，驱动、应用编写很多时候都需要使用链表。所以对链表必须掌握。掌握到：会自己定义结构体来实现链表、会写链表的节点插入（前插、后插）、节点删除、节点查找、节点遍历等。（至于像逆序这些很少用）

哈希表不是很常用，一般不需要自己写实现，而直接使用别人实现的哈希表表较多。对我们来说，最重要的是明白哈希表的原理、从而知道哈希表的特点，从而知道什么时候该使用哈希表，当看到别人用了哈希表的时候要明白别人为什么要用哈希表、合适不合适？有没有更好的选择？

二叉树、图不用深究。

### 1.8.2.为什么需要复杂数据结构

因为现实中的实际问题是多种多样的，问题的复杂度不同，所以需要解决问题的算法和数据结构也不同。所以当你处理什么复杂度较高的问题，就去研究针对性解决的数据结构和算法。

### 1.8.3.数据结构和算法（可以用程序实现的代码）的关系

数据结构的发明都是为了配合一定的算法；算法是为了处理具体问题，算法的实现依赖于相应的数据结构。

# 二、C语言位操作

## 2.1位操作符

### 2.1.1.位与&

(1)注意：位与符号是一个&，两个&&是逻辑与。

(2)真值表：1&0=0 1&1=1 0&0=0 0&1=0

(3)从真值表可以看出：位与操作的特点是，只有1和1位于结果为1，其余全是0.

(4)位与和逻辑与的区别：位与时两个操作数是按照二进制位彼次对应位相与的，逻辑与是两个操作数作为整体来相与的。（举例：0xAA&0xF0=0xA0, 0xAA && 0xF0=1）

### 2.1.2.位或|

(1)注意：位或符号是一个|，两个||是逻辑或。

(2)真值表：1|0=1 1|1=1 0|0=0 0|1=1

(3)从真值表可以看出：位或操作的特点是：只有2个0相位或才能得到0，只要有1个1结果就一定是1.

(4)位或和逻辑或的区别：位或时两个操作数是按照二进制位彼次对应位相与的，逻辑或是两个操作数作为整体来相或的。

### 2.1.3.位取反~

(1)注意：C语言中位取反是~，C语言中的逻辑取反是!

(2)按位取反是将操作数的二进制位逐个按位取反（1变成0，0变成1）；而逻辑取反是真（在C语言中只要不是0的任何数都是真）变成假（在C语言中只有0表示假）、假变成真。

任何非0的数被按逻辑取反再取反就会得到1；

任何非0的数被按位取反再取反就会得到他自己；

### 2.1.4.位异或^

(1)位异或真值表：1^1=0 0^0=0 1^0=1 0^1=1

(2)位异或的特点：2个数如果相等结果为0，不等结果为1。记忆方法：异或就是相异就或操作起来。

位与、位或、位异或的特点总结：

位与：（任何数，其实就是1或者0）与1位与无变化，与0位与变成0

位或：（任何数，其实就是1或者0）与1位或变成1，与0位或无变化

位异或：（任何数，其实就是1或者0）与1位异或会取反，与0位异或无变化

### 2.1.5.左移位<< 与右移位>>

C语言的移位要取决于数据类型。

对于无符号数，左移时右侧补0（相当于逻辑移位）

对于无符号数，右移时左侧补0（相当于逻辑移位）

对于有符号数，左移时右侧补0（叫算术移位，相当于逻辑移位）

对于有符号数，右移时左侧补符号位（如果正数就补0，负数就补1，叫算术移位）

嵌入式中研究的移位，以及使用的移位都是无符号数

## 2.2位与位或位异或在操作寄存器时的特殊作用

### 2.2.1寄存器操作的要求（特定位改变而不影响其他位）

(1)ARM是内存与IO统一编址的，ARM中有很多内部外设，SoC中CPU通过向这些内部外设的寄存器写入一些特定的值来操控这个内部外设，进而操控硬件动作。所以可以说：读写寄存器就是操控硬件。

(2)寄存器的特点是按位进行规划和使用。但是寄存器的读写却是整体32位一起进行的（也就是说你只想修改bit5～bit7是不行的，必须整体32bit全部写入）

(3)寄存器操作要求就是：在设定特定位时不能影响其他位。

(4)如何做到？答案是：读-改-写三部曲。读改写的操作理念，就是：当我想改变一个寄存器中某些特定位时，我不会直接去给他写，我会先读出寄存器整体原来的值，然后在这个基础上修改我想要修改的特定位，再将修改后的值整体写入寄存器。这样达到的效果是：在不影响其他位原来值的情况下，我关心的位的值已经被修改了。

### 2.2.2特定位清零用&

(1)回顾上节讲的位与操作的特点：（任何数，其实就是1或者0）与1位与无变化，与0位与变成0

(2)如果希望将一个寄存器的某些特定位变成0而不影响其他位，可以构造一个合适的1和0组成的数和这个寄存器原来的值进行位与操作，就可以将特定位清零。

(3)举例：假设原来32位寄存器中的值为：0xAAAAAAAA，我们希望将bit8～bit15清零而其他位不变，可以将这个数与0xFFFF00FF进行位与即可。

### 2.2.3特定位置1用|

(1)回顾上节讲的位或操作的特点：任何数，其实就是1或者0）与1位或变成1，与0位或无变化

(2)操作手法和刚才讲的位与是类似的。我们要构造这样一个数：要置1的特定位为1，其他位为0，然后将这个数与原来的数进行位或即可。

### 2.2.4特定位取反用^

(1)回顾上节讲的位异或操作的特点：（任何数，其实就是1或者0）与1位异或会取反，与0位异或无变化

(2)操作手法和刚才讲的位与是类似的。我们要构造这样一个数：要取反的特定位为1，其他位为0，然后将这个数与原来的数进行位异或即可。

## 2.3如何用位运算构建特定二进制数

### 2.3.1.寄存器位操作经常需要特定位给特定值

(1)从上节可知，对寄存器特定位进行置1或者清0或者取反，关键性的难点在于要事先构建一个特别的数，这个数和原来的值进行位与、位或、位异或等操作，即可达到我们对寄存器操作的要求。

(2)解法1：用工具软件或者计算器或者自己大脑计算，直接给出完整的32位特定数。

优势：可以完成工作，难度也不大，操作起来也不是太麻烦。

劣势：依赖工具，而且不直观，读程序的人不容易理解。

评价：凑活能用，但是不好用，应该被更好用的方法替代。

(2)解法2：自己写代码用位操作符号（主要是移位和位取反）来构建这个特定的二进制数

### 2.3.2.使用移位获取特定位为1的二进制数

(1)最简单的就是用移位来获取一个特定位为1的二进制数。譬如我们需要一个bit3～bit7为1（隐含意思就是其他位全部为0）的二进制数，可以这样：(0x1f<<3)

(2)更难一点的要求：获取bit3～bit7为1，同时bit23～bit25为1，其余位为0的数：((0x1f<<3) | (0x7<<23))

一个1 0x1

两个1 0x3

三个1 0x7

四个1 0xf

五个1 0x1f

六个1 0x3f

七个1 0x7f

八个1 0xff

### 2.3.3.再结合位取反获取特定位为0的二进制数

(1)这次我们要获取bit4～bit10为0，其余位全部为1的数。怎么做？

(2)利用上面讲的方法就可以：(0xf<<0)|(0x1fffff<<11)

但是问题是：连续为1的位数太多了，这个数字本身就很难构造，所以这种方法的优势损失掉了。

(3)这种特定位（比较少）为0而其余位（大部分）为1的数，不适合用很多个连续1左移的方式来构造，适合左移加位取反的方式来构造。

(2)思路是：先试图构造出这个数的位相反数，再取反得到这个数。（譬如本例中要构造的数bit4～bit10为0其余位为1，那我们就先构造一个bit4～bit10为1，其余位为0的数，然后对这个数按位取反即可） ~(0x7f<<4)

### 2.3.4.总结：位与、位或结合特定二进制数即可完成寄存器位操作需求

(1)如果你要的这个数比较少位为1，大部分位为0，则可以通过连续很多个1左移n位得到。

(2)如果你想要的数是比较少位为0，大部分位为1，则可以通过先构建其位反数，然后再位取反来得到。

(3)如果你想要的数中连续1（连续0）的部分不止1个，那么可以通过多段分别构造，然后再彼此位与即可。这时候因为参与位或运算的各个数为1的位是不重复的，所以这时候的位或其实相当于几个数的叠加。

## 2.4位操作实战

(1)给定一个整型数a，设置a的bit3，保证其他位不变。

a |= (1<<3)

(2)给定一个整形数a，设置a的bit3~bit7，保持其他位不变

a |= (0x1f<<3)

(3)给定一个整型数a，清除a的bit15，保证其他位不变。

a &= ~(1<<15)

(4)给定一个整形数a，清除a的bit15~bit23，保持其他位不变。

a &= ~(0x1ff<<15)

(5)给定一个整形数a，取出a的bit3~bit8。

a &= (0x3f<<3)

a >>= 3

(6)给一个寄存器的bit7～bit17赋值937

a &= ~(0x7ff<<7)

a |= (937<<7)

(7)给一个寄存器的bit7～bit17中的值加17

b = ((a & (0x7ff<<7))>>7 + 17)<<7

a &= ~(0x7ff<<7)

a |= b

(8)给一个寄存器的bit7～bit17赋值937，同时给bit21～bit25赋值17.

a &= ~((0x7ff<<7) | (0x1f<<21))

a |= ((937<<7) | (17<<21))

(9)用宏定义来完成位运算

截取变量的部分连续位：

#define GETBITS(x, n, m) ((x & (~(~(0U)<<(m-n+1)))<<(n-1)) >> (n-1))

分析：这个题目相当于我们(5)中做的事情，只不过要用宏来实现。

这个题目相当于是要把x的bit(n-1)到bit(m-1)取出来

复杂宏怎么分析：

((x & ~(~(0U)<<(m-n+1))<<(n-1)) >> (n-1))

第一步，先分清楚这个复杂宏分为几部分：2部分

(x & ~(~(0U)<<(m-n+1))<<(n-1)) >> (n-1)

分析为什么要>>(n-1)，相当于是我们(5)中的第二步

第二步，继续解析剩下的：又分为2部分

x & ~(~(0U)<<(m-n+1))<<(n-1)

分析为什么要&，相当于我们(5)中的第一步

第三步，继续分析剩下的：

~ (~(0U)<<(m-n+1)) << (n-1)

这个分析时要搞清楚第2坨到底应该先左边取反再右边<<还是先右边<<再左边取反。

解法：第一，查C语言优先级表；第二，自己实际写个代码测试。

说明这个式子应该是 ~(~(0U)<<(m-n+1)) << (n-1) ，这就又分为2部分了

# 三、指针才是C的精髓

## 3.1指针到底是什么

### 3.1.1.指针变量和普通变量的区别

首先必须非常明确：指针的实质就是个变量，它跟普通变量没有任何本质区别。指针完整的名字叫指针变量，简称指针。

### 3.1.2.为什么需要指针

(1)指针的出现是为了实现间接访问。在汇编中都有间接访问（CPU访问内存通过CPU中的寄存器写入内存地址来实现间接访问,譬如：ldr r0, [r1] 将CPU中的r1寄存器中保存的DDR内存地址处所保存在DDR内存中的值赋值给r0寄存器），其实就是CPU的寻址方式中的间接寻址。

(2)间接访问（CPU的间接寻址）是CPU设计时决定的，这个决定了汇编语言必须能够实现间接寻址，又决定了汇编之上的C语言也必须实现间接访寻址。

(3)高级语言如Java、C#等没有指针，那么他们怎么实现间接访问？答案是语言本身帮我们封装了。

### 3.1.3.指针使用三部曲：定义指针变量、关联指针变量、解引用

(1)当我们int \*p 定义一个指针变量p时，因为p是局部变量，所以也遵循C语言局部变量的一般规律（定义局部变量并且未初始化，则值是随机的），所以此时p变量中存储的是一个随机的数字。

(2)此时如果我们解引用p，则相当于我们访问了这个随机数字为地址的内存空间。那这个空间到底能不能访问不知道（也许可以也许不行），所以如果直接定义指针变量未绑定有效地址就去解引用几乎必死无疑。

(3)定义一个指针变量，不经绑定有效地址就去解引用，就好象拿一个上了镗的枪在四面八方中随意开了一枪。

(4)指针绑定的意义就在于：让指针指向一个可以访问、应该访问的地方（就好象拿着枪瞄准目标的过程一样），指针的解引用是为了间接访问目标变量（就好象开枪是为了打中目标一样）



## 3.2指针带来的一些符号的理解

我们写的代码是给编译器看的，代码要想达到你想象的结果，就必需要编译器对你的代码的理解和你自己对代码的理解一样。编译器理解代码就是理解的符号，所以我们要正确理解C语言中的符号，才能像编译器一样思考程序、理解代码。

### 3.2.1.星号\*

(1)C语言中\*可以表示乘号，也可以表示指针符号。这两个用法是毫无关联的，只是恰好用了同一个符号而已。

(2)\*在用于指针相关功能的是后续有2种用法：第一种是指针定义时，\*结合前面的类型用于表明要定义的指针的类型；第二种功能是指针解引用，解引用时\*p表示p指向的变量本身。

### 3.2.2.取地址符&

取地址符使用时直接加在一个变量的前面，然后取地址符和变量加起来构成一个新的符号，这个符号表示这个变量的地址。

### 3.2.3.指针定义并未初始化、与指针定义然后赋值的区别

(1)指针定义时可以初始化，指针的初始化其实就是给指针变量初值（跟普通变量的初始化没有任何本质区别）。

(2)指针变量定义同时初始化的格式是：int a = 32; int \*p = &a;

(3)不初始化时指针变量先定义再赋值：int a = 32; int \*p; p = &a;

不正确 ---- \*p = &a;

### 3.2.4.左值与右值

(1)放在赋值运算符左边的就叫左值，右边的就叫右值。所以赋值操作其实就是：左值 = 右值;

(2)当一个变量做左值时，编译器认为这个变量符号的真实含义是这个变量所对应的那个内存空间；当一个变量做右值时，编译器认为这个变量符号的真实含义是这个变量的值，也就是这个变量所对应的内存空间中存储的那个数。

(3)左值与右值的区别，就好象现实生活中“家”这个字的含义。譬如“我回家了”，这里面的家指的是你家的房子（类似于左值）；但是说“家比事业重要”，这时候的家指的是家人（家人就是住在家所对应的那个房子里面的人，类似于右值）



## 3.3野指针问题

### 3.3.1.神马是野指针？哪里来的？有什么危害？

我的理解：野指针就是定义了指针没有给指针赋值。

(1)野指针，就是指针指向的位置是不可知的（随机的、不正确的、没有明确限制的）

(2)野指针很可能触发运行时段错误（Sgmentation fault）

(3)因为指针变量在定义时如果未初始化，值也是随机的。指针变量的值其实就是别的变量（指针所指向的那个变量）的地址，所以意味着这个指针指向了一个地址是不确定的变量，这时候去解引用就是去访问这个地址不确定的变量，所以结果是不可知的。

(4)野指针因为指向地址是不可预知的，所以有3种情况：第一种是指向不可访问（操作系统不允许访问的敏感地址，譬如内核空间）的地址，结果是触发段错误，这种情况算好的了，因为编译器会报错；第二种是指向一个可用的、而且没有什么特别意义的空间（譬如我们曾经使用过但是已经不用的栈空间或堆空间），这时候程序运行不会出错，也不会对当前程序造成损害，这种情况下会掩盖你的程序错误，让你以为程序没问题，其实是有问题的；第三种情况就是指向了一个可用的空间，而且这个空间其实在程序中正在被使用（譬如说是程序的一个变量x），那么野指针的解引用就会刚好修改这个变量x的值，导致这个变量莫名其妙的被改变，程序出现离奇的错误。一般最终都会导致程序崩溃，或者数据被损害。这种危害是最大的。

(5)指针变量如果是局部变量，则分配在栈上，本身遵从栈的规律（反复使用，使用完不擦出，所以是脏的，本次在栈上分配到的变量的默认值是上次这个栈空间被使用时余留下来的值），就决定了栈的使用多少会影响这个默认值。因此野指针的值是有一定规律不是完全随机的，但是这个值的规律对我们没意义。因为不管落在上面的野指针是3种情况的哪一种，都不是我们想看到的。

## 3.3.2.怎么避免野指针？

(1)野指针的错误来源就是指针定义了以后没有初始化，也没有赋值（总之就是指针没有明确的指向一个可用的内存空间），然后去解引用。

(2)知道了野指针产生的原因，避免方法就出来了：在指针的解引用之前，一定确保指针指向一个绝对可用的空间。

(3)常规的做法是：

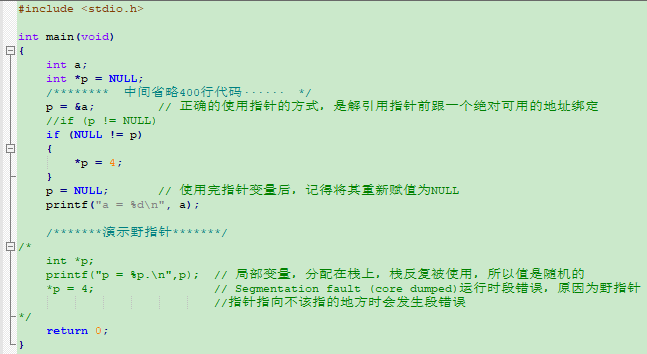
第一点：定义指针时，同时初始化为NULL

第二点：在指针使用之前，将其赋值绑定给一个可用地址空间

第三点：在指针解引用之前，先去判断这个指针是不是NULL

第四点：指针使用完之后，将其赋值为NULL

(4)野指针的防治方案4点绝对可行，但是略显麻烦。很多人懒得这么做，实战中是怎么处理？在中小型程序中，自己水平可以把握的情况下，不必严格参照这个标准；但是在大型程序中，或者自己水平感觉不好把握时，建议严格参照这个方法。



### 3.3.3.NULL到底是什么？

(1)NULL在C/C++中定义为：

#ifdefine \_cplusplus //定义这个符号就表示当前是C++环境

#define NULL 0 //在C++中NULL就是0

#else

#define NULL (void \*)0 //在C中NULL是强制类型转换为void \*的0

#endif

(2)在C语言中，int \*p，你可以p = (int \*)0，但是不可以p = 0，因为类型不同

(3)所以NULL的实质其实就是地址0，然后我们给指针赋初值为NULL，其实就是让指针指向0地址处。为什么指向0地址处？有2点原因：第一层原因是0地址处作为一个特殊地址（我们认为指针指向这里就表示指针没有被初始化，就表示野指针）；第二层原因是这个0地址在一般的操作系统中都是不可被访问的，如果不按规矩（不检查是否等于NULL就去解引用）写代码直接去解引用就会触发段错误，编译器会报错嘀。(指针指向这个敏感地址没有问题，只要不解引用就ok)

## 3.4 const关键字与指针

### 3.4.1. const修饰指针的4种形式

(1)const关键字，在C语言中用来修饰变量，表示这个变量是常量。

(2)const修饰指针有4种形式，区分清楚这4种即可全部理解const和指针。

第一种：const int \*p; //const修饰p指向的变量

第二种：int const \*p; //const修饰p指向的变量

第三种：int \* const p; //const修饰指针变量p

第四种：const int \* const p; //const即修饰p指向的变量也修饰指针变量p

(3)关于指针变量的理解，主要涉及到2个变量：第一个是指针变量p本身，第二个是p指向的那个变量(\*p)。一个const关键字只能修饰一个变量，所以弄清楚这4个表达式的关键就是搞清楚const放在某个位置是修饰谁。

### 3.4.2.const修饰的变量真的不能改吗？

(1)课题练习说明：const修饰的变量其实是可以改的（前提是gcc环境下）。

(2)在某些单片机环境下，const修饰的变量是不可以修改的。const修饰的变量能不能直接被修改，取决于具体的环境，C语言本身并没有完全严格一致的要求。

(3)在gcc中，const关键字是通过编译器在编译的时候执行检查来确保实现的（也就是说const类型的变量不能改是编译错误，不是运行时错误），所以我们只要想办法骗过编译器(const )，就可以修改const定义的常量，而运行时不会报错。

(4)更深一层的原因，是因为gcc把const类型的常量也放在了.data段，其实和普通的全局变量放在.data段是一样实现的，只是通过编译器认定这个变量是const的，运行时并没有标记const标志，所以只要骗过编译器就可以修改了。

### 

### 3.4.3.const究竟应该怎么用？

const是在编译器中实现的，编译时检查，并非不能骗过。所以在C语言中使用const，就好像是一种道德约束而非法律约束，所以大家使用const时更多是传递一种信息，就是告诉编译器、也告诉读程序的人，这个变量是不应该也不必被修改的。

## 3.5 深入学习下数组

### 3.5.1.从内存角度来理解数组

(1)从内存角度讲，数组变量就是一次分配多个变量，而且这多个变量在内存中的存储单元是依次相连接的。

(2)我们分开定义多个变量（譬如int a, b, c, d）和和一次定义一个数组（int a[4]）这两种定义方法相同点是都定义了4个int型变量，而且这4个变量都是独立的、单个使用的；不同点是单独定义时a、b、c、d在内存中的地址不一定相连，但是定义成数组后，数组中的4个元素地址肯定是依次相连的。

(3)数组中多个变量虽然必须单独访问，但是因为他们的地址彼此相连，因此很适合用指针来操，因此数组和指针天生就有一种羁绊。

### 3.5.2.从编译器角度来理解数组

(1)从编译器角度来讲，数组变量也是变量，和普通变量和指针变量并没有本质不同。变量的本质就是一个地址，这个地址在编译器中决定具体数值，具体数值和变量名绑定，变量类型决定这个地址的延续长度。

(2)搞清楚：变量、变量名、变量类型这三个概念的具体含义，很多问题都清楚了。

### 3.5.3.数组中几个关键符号（a a[0] &a &a[0]）的理解（假如是int a[10]）

(1)这4个符号搞清楚了，数组相关的很多问题就都有答案了。理解这些符号的时候要和左值右值结合起来，也就是搞清楚每个符号分别做左值和右值时的不同含义。

(2)a就是数组名。a做左值表示整个数组所有空间（10\*4=40字节），但因为C语言规定数组操作时要独立单个操作，不能整体操作数组，所以a不能做左值；a做右值表示数组首元素（数组的第1个元素，也就是a[0]）的首地址（首地址就是起始地址，就是4个字节中最开始第一个字节的地址）。a做右值等于&a[0]。

(3)a[0]表示数组的首元素，也就是数组的第1个元素。a[0]做左值时表示首元素对应的内存空间（连续4个字节）；a[0]做右值时表示数组第1个元素的值（也就是数组第1个元素对应的内存空间中存储的那个数）

(4)&a就是数组名a取地址，字面意思来看就应该是数组的地址。&a不能做左值（&a实质是一个常量，不是变量，因此不能赋值，所以自然不能做左值。）；&a做右值时表示整个数组的首地址。

(5)&a[0]字面自已就是数组第1个元素的首地址（搞清楚[]和&的优先级，[]的优先级要高于&，所以a先和[]结合再取地址）。&a[0]不能做左值（&a[0]实质是一个常量，不是变量，因此不能赋值，所以自然不能做左值。）；做右值时表示数组首元素地址。&a[0]做右值等于a。

解释：为什么数组的地址是常量？因为数组是编译器在内存中自动分配的。当我们每次执行程序时，运行时都会帮我们分配一块内存给这个数组，只要完成了分配，这个数组的地址就定好了，本次程序运行直到终止都无法再改了。那么我们在程序中只能通过&a来获取这个分配的地址，却不能去赋值运算符修改它。

总结：

(1)a 不能做左值；做右值相当于&a[0]做右值，是数组首元素地址

a[0] 做左值是数组首元素存储空间；做右值是数组首元素的值

&a 不能做左值；做右值是整个数组的首地址（值等于首元素地址而已，意义不同）

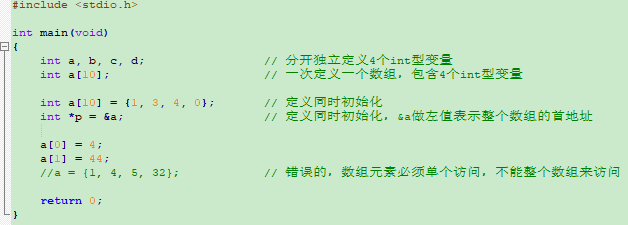
&a[0] 不能做左值；做右值是数组首元素地址

(2)&a和a、&a[0]做右值时的区别：&a是整个数组的首地址，而a和&a[0]是数组首元素的首地址。虽然这两个在数值上是相等的，但是意义不同。意义不相同会导致他们在参与运算的时候有不同的表现。

(3)a和&a[0]做右值时数值和意义完全相同，完全可以互相替代。

(4)&a和&a[0]是常量，不能做左值。

(5)a做左值代表整个数组所有空间，所以不能做左值。



## 3.6 指针和数组的天生羁绊

### 3.6.1.以指针方式来访问数组元素

(1)数组元素使用时不能整体访问，只能单个访问。访问方式有2种：数组形式和指针形式。

(2)数组格式访问数组元素是：数组名[下标];

(3)指针方式访问数组元素是：\*(指针+偏移量); \*(数组名+偏移量);

如果指针是数组首元素地址（a或者&a[0]），那么偏移量就是下标；指针也可以不是首元素地址而是其他哪个元素的地址，这时候偏移量就要考虑叠加了。

(4)数组下标方式和指针方式均可以访问数组元素，两者的实质其实是一样的。在编译器内部都是用指针方式来访问数组元素的，数组下标方式只是编译器提供给编程者一种壳（语法糖）而已。所以用指针方式来访问数组才是本质的方法。

### 3.6.2.从内存角度理解指针访问数组的实质

(1)数组的特点就是：数组中各个元素的地址是依次相连的，而且数组还有一个很大的特点（其实也是数组的一个限制）就是数组中各个元素的类型相同。类型相同就决定了每个数组元素占几个字节是相同的（譬如int数组每个元素都占4字节，没有例外）。

(2)数组中的元素其实就是地址相连接、占地大小相同的一串内存空间。这两个特点就决定了只要知道数组中一个元素的地址，就可以很容易推算出其他元素的地址。

### 3.6.3.指针和数组类型的匹配问题

(1)int \*p; int a[5]; p = a; //类型匹配，a相当于&a[0]

(2)int \*p; int a[5]; p = &a; //类型不匹配

p是int \*，是int指针类型，而&a是整个数组的指针，也就是数组指针类型，所以不匹配

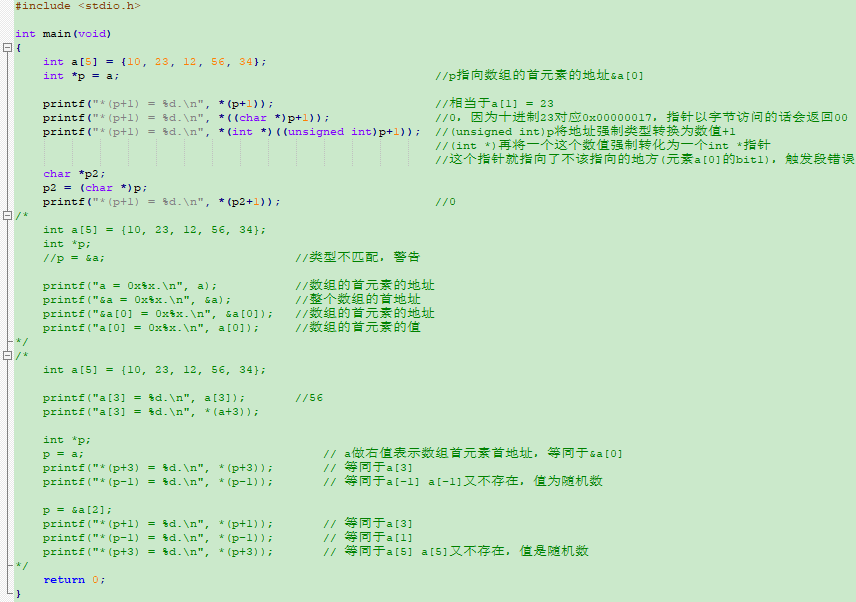
(3)&a和a、&a[0]从数值来看是相等的，但是意义来看就不同了。从意义来看，a和&a[0]是数组首元素地址，而&a是整个数组的首地址；从类型来看，a和&a[0]是元素的指针，也就是int \*类型的；而&a是数组指针，是int (\*)[5]类型。

### 3.6.4.总结：指针类型决定了指针如何参与运算

(1)指针参与运算时，因为指针变量本身存储的数值是表示地址的，所以运算也是地址的运算。

(2)指针参与运算的特点是，指针变量+1，并不是真的＋1，而是加1\*sizeof(指针类型)。如果是int \*指针，则+1就实际表示地址+4即1\*sizeof(int)；如果是char \*指针，则+1就实际表示地址+1即1\*sizeof(char)；如果是double \*指针，则+1就实际表示地址+8即1\*sizeof(double)。（隐含知识：地址是以字节为单位的）

(3)指针变量+1时实际不是加1而是加1\*sizeof(指针类型)，主要原因是希望指针+1后刚好指向下一个元素(而不希望错位)。



## 3.7 指针与强制类型转换

### 3.7.1.变量的数据类型的含义

(1)所有的类型的数据存储在内存中，都是按照二进制格式存储的。所以内存中只知道1和0，不知道int的还是float的还是其他类型。

(2)int、short、char等属于整型，他们的存储方式（数转换成二进制往内存中存放的方式）是相同的，只是内存格子大小不同（所以这几种整型就彼此叫二进制兼容格式）；而float、double的存储方式彼此不同，和整型更不同。

(3)int a = 5; 时，编译器给a分配4字节空间，并且将5按照int类型的存储方式转换成二进制存到a所对应的内存空间中去（a做左值的）；我们printf去打印a的时候（a此时做右值），printf内部的vsprintf函数会按照格式化字符串（就是printf传参的第一个字符串参数中的%d之类的东西）所代表的类型去解析a所对应的内存空间，解析出的值用来输出。也就是说，存进去时是按照这个变量本身的数据类型来存储的（譬如本例中a为int所以按照int格式来存储）；但是取出来时是按照printf中%d之类的格式化字符串的格式来提取的。此时虽然a所代表的内存空间中的1010序列并没有变（内存是没被修改的）但是怎么理解（怎么把这些1010转成数字）就不一定了。譬如我们用%d来解析，那么还是按照int格式解析则值自然还是5；但是如果用%f来解析，则printf就以为a对应的内存空间存储的是一个float类型的数，会按照float类型来解析，值自然是很奇怪的一个数字了。

总结：C语言中的数据类型的本质，就是决定了这个数在内存中是怎么存储的问题，也就是决定了这个数如何转成二进制的问题。一定要记住的一点是内存中只是存储1010的序列，而不管这些1010是怎么解析。所以要求我们平时数据类型不能瞎胡乱搞。

分析几个题目：

\* 按照int类型存却按照float类型取 一定会出错

\* 按照int类型存却按照char类型取 有可能出错也有可能不出错

\* 按照short类型存却按照int类型取 有可能出错也有可能不出错

\* 按照float类型存却按照double取 一定会出错

### 3.7.2.指针的数据类型的含义

(1)指针的本质是：变量，指针就是指针变量

(2)一个指针涉及2个变量：一个是指针变量自己本身，一个是指针变量指向的那个变量

(3)int \*p; 定义指针变量时，p（指针变量本身）是int \*类型， \*p（指向的那个变量）是int类型的。

(4)int \*类型说白了就是指针类型，只要是指针类型就都是4个字节，解析方式都是按照地址的方式来解析（意思是里面存的32个二进制位合起来表示一个内存地址）的。结论就是：所有指针类型（不算是int \*还是char \*还是int (\*)[5]）的解析方式都是相同的，都是地址。

(5)对于指针所指向的那个变量来说，指针的类型就很重要了。指针所指向的那个变量的类型（它所对应的内存空间的解析方法）要取决于指针类型。譬如指针是int \*的，那么指针所指向的变量就是int类型的。

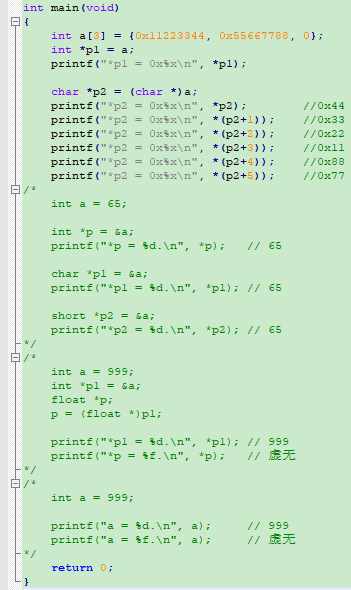
### 3.7.3.指针数据类型转换实例分析1（int \* -> char \*）

(1)int和char类型都是整型的，类型兼容的。所以互转的时候有可能出错有可能对。

(2)int和char的不同在于char只有1个字节而int有4个字节，所以int的范围比char大。在char所表示的范围之内int和char是可以互转的不会出错；但是超过了char的范围后，char转成int不会错（向大方向转就不会出错，就好比拿小瓶子的水往大瓶子倒不会漏掉），而从int到char转就会出错（就好像拿大瓶子的水往小瓶子倒一样）

### 3.7.4.指针数据类型转换实例分析2(int \* -> float \*)

因为int和float的解析方式是不兼容的，所以int \*转成float \*再去访问绝对会出错。



## 3.8 指针、数组与sizeof运算符

### 3.8.1. sizeof运算符

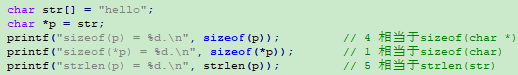
(1)sizeof是C语言中的一个运算符（sizeof不是函数，虽然用法很像函数），sizeof的作用是用来返回()里边的变量或者数据类型占用的内存字节数。

(2)sizeof存在的价值？主要是因为在不同平台下各种数据类型所占的字节数不尽相同（譬如int在32位系统中为4字节，在16位系统中为2字节...）。所以程序中需要使用sizeof来判断当前变量/数据类型在当前环境下占几个字节。

### 3.8.2. char str[] = ”hello”; sizeof(str) sizeof(str[0]) strlen(str)



### 3.8.3. char \*p = str; sizeof(p) sizeof(\*p) strlen(p)



(1)32位系统中所有指针的长度都是4，不管是什么类型的指针。

(2)strlen是一个C库函数，用来返回一个字符串的长度（注意，字符串的长度是不计算字符串末尾的'\0'的）。一定要注意strlen接收的参数必须是一个字符串（字符串的特征是以'\0'结尾）。

### 3.8.4.int n=10; sizeof(n)

sizeof测试一个变量本身，和sizeof测试这个变量的类型，结果是一样的。

### 3.8.5.int b[100]; sizeof(b)

sizeof(数组名)的时候，数组名不做左值也不做右值，纯粹就是数组名的含义。那么sizeof(数组名)实际返回的是整个数组所占用内存空间（以字节为单位的）。

3.8.6. void fun(int b[100])

{

sizeof(b) // 4

}

(1)函数传参，形参可以用数组的

(2)函数形参是数组时，实际传递的不是整个数组，而是数组的首元素地址。也就是说函数传参用数组来传，实际相当于传递的是指针（指针指向数组的首元素地址）。



### 3.8.7.通过sizeof获取数组元素个数的技巧

int a[56];

int b = sizeof(a) / sizeof(a[0]); // 整个数组字节数/数组中一个元素的字节数

printf("b = %d.\n", b); // 结果应该是数组的元素个数

### 3.8.8. #define和typedef的区别

#define dpChar char \*

typedef char \*tpChar;

dpChar p1, p2; sizeof(p1) sizeof(p2)

tpChar p3, p4; sizeof(p3) sizeof(p4)

dpChar p1, p2; //展开：char \*p1, p2; 相当于char \*p1, char p2;

tpChar p3, p4; // 等价于：char \*p3, char \*p4;

## 3.9 指针与函数传参

### 3.9.1.普通变量作为函数形参

(1)函数传参时，普通变量作为参数时，形参和实参名字可以相同也可以不同，实际上都是用实参来替代相对应的形参的。

(2)在子函数内部，形参的值等于实参。原因是函数调用时把实参的值赋值给了形参。

(3)这就是很多书上写的“传值调用”（相当于实参做右值，形参做左值）

int a = 4;

printf("&a = %p.\n", &a);

func1(a);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void func1(int b)

{

printf("b = %d.\n", b); // 在函数内部，形参b的值等于实参a

printf("in func1, &b = %p.\n", &b);

}

//&a和&b不同，说明a和b不是同一个变量（在内存中a和b是独立的2个内存空间）但是a和b是有关联的，实际上b是a赋值得到的。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 3.9.2.数组作为函数形参

(1)数组作为形参传参时，实际传递的不是整个数组，而是数组的首元素地址（也就是整个数组的首地址。因为传参时是传值，所以首元素地址和数组的首地址这两个没区别）。所以在子函数内部，传进来的数组名就等于是一个指向数组首元素的指针。所以sizeof得到的是4。

(2)在子函数内传参得到的数组首元素首地址，和外面得到的首元素首地址是相同的。很多人把这种特性叫做“传址调用”（所谓的传址调用就是调用子函数时传了地址（也就是指针），此时可以通过传进去的地址来访问实参。）

(3)数组作为函数参数时，[]里的数组是可有可无的。为什么？因为数组名做形参传递的实际只是个指针，根本没有数组长度这个信息。

int a[5];

printf("a = %p.\n", a);

func2(a);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void func2(int a[])

{

printf("sizeof(a) = %d.\n", sizeof(a)); //4

printf("in func2, a = %p.\n", a);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 3.9.3.指针作为函数形参

和数组作为函数形参是一样的，这就好像指针方式访问数组元素和数组方式访问数组元素的结果一样是一样的。

int a[5];

printf("a = %p.\n", a);

func3(a);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void func3(int \*a)

{

printf("sizeof(a) = %d.\n", sizeof(a));

printf("in func2, a = %p.\n", a);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 3.9.4.结构体变量和结构体变量指针作为函数形参

(1)结构体变量作为函数形参的时候，实际上和普通变量（类似于int之类的）传参时的表现是一模一样的。所以说结构体变量其实也是普通变量而已。

(2)因为结构体一般都很大，所以如果直接用结构体变量进行传参，那么函数调用效率就会很低。（因为在函数传参时需要将实参赋值给形参，所以当传参的变量越大调用效率就会越低。）怎么解决？思路只有一个那就是不要传变量了，改传变量的指针（地址）进去。

(3)结构体因为自身太大，所以传参应该用结构体指针来传（但是程序员可以自己决定，你非要传结构体变量过去C语言也是允许的，只是效率低了）；回想一下数组，为什么C语言设计时数组传参默认是传的数组的首元素地址而不是整个数组？因为数组一般也很大，与其传值不如传址，进而直接操作实参。

struct A

{

char a; // 结构体变量对齐问题

int b; // 因为要对齐存放，所以大小是8

};

/\*

int main()

{

struct A a =

{

.a = 4,

.b = 5555,

};

printf("sizeof(a) = %d.\n", sizeof(a)); // 8

printf("&a = %p.\n", &a); // 结构体变量a的地址

printf("a.b = %d.\n", a.b); // 5555

func4(a);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void func4(struct A a1)

{

printf("sizeof(a1) = %d.\n", sizeof(a1)); // 8

printf("&a1 = %p.\n", &a1); // 结构体变量a1的地址

printf("a1.b = %d.\n", a1.b); // 5555

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

\*/

int main()

{

struct A a =

{

.a = 4,

.b = 5555,

};

printf("sizeof(a) = %d.\n", sizeof(a)); // 8 结构体变量内存对齐，占8字节空间

//printf("sizeof(\*a) = %d.\n", sizeof(\*a)); // error，a本身就是一个普通结构体变量，变量的解引用是什么鬼。

printf("&a = %p.\n", &a); // 结构体变量a的地址

//printf("a = %p.\n", a); // 结构体变量a中保存的值按照%p解析而已

printf("a.b = %d.\n\n", a.b); // 5555

func5(&a);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void func5(struct A \*a1)

{

//printf("sizeof(a1) = %d.\n", sizeof(a1)); // 32位系统指针变量4个字节

printf("sizeof(\*a1) = %d.\n", sizeof(\*a1)); // 8

//printf("&a1 = %p.\n", &a1); // 二重指针，指向结构体指针地址的地址

printf("a1 = %p.\n", a1); // 结构体变量a的地址

printf("a1->b = %d.\n", a1->b); // 5555

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 3.9.5.传值调用与传址调用

(1)传值调用描述的是这样一种现象：x和y作为实参，自己并没有真身进入swap1函数内部，而只是拷贝了一份自己的副本（副本具有和自己一样的值，但是是不同的变量）进子函数swap1，然后我们在子函数swap1中交换的实际是副本而不是x、y真身。所以在swap1内部确实是交换了，但是到外部的x和y根本没有受影响。

int x = 3, y = 5;

swap1(x, y);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void swap1(int a, int b)

{

int tmp;

tmp = a;

a = b;

b = tmp;

printf("in swap1, a = %d, b = %d.\n", a, b);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

printf("x = %d, y = %d.\n", x, y); // x=3,y=5，交换失败

(2)在swap2中x和y真的被改变了（但是x和y真身还是没有进入swap2函数内，而是swap2函数内部跑出来把外面的x和y真身给改了）。实际上实参x和y永远无法真身进入子函数内部（进去的只能是一份拷贝），但是在swap2我们把x和y的地址传进去给子函数，于是在子函数内可以通过指针解引用方式从函数内部访问到外部的x和y真身，从而改变x和y。

int x = 3, y = 5;

swap2(&x, &y);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void swap2(int \*a, int \*b)

{

int tmp;

tmp = \*a;

\*a = \*b;

\*b = tmp;

printf("in swap2, \*a = %d, \*b = %d.\n", \*a, \*b);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

printf("x = %d, y = %d.\n", x, y); // x=5,y=3，交换成功

## 3.10 输入型参数和输出型参数

### 3.10.1.函数为什么需要形参和返回值

(1)函数名是一个符号，表示整个函数代码段的首地址，实质是个指针变量，所以在程序中使用到函数名时都是当地址用的，用来调用这个函数的。

(2)函数体是函数的关键，有{}括起来，包含很多句代码，函数体就是函数实际做的工作。

(3)形参列表和返回值。形参是函数的输入部分，返回值就是函数的输出部分。对函数最好的理解就是把函数看成是一个加工机器（程序其实就是数据加工器），形参列表就是这个机器的原材料输入端，而返回值就是机器的成品输出端。

(4)其实如果没有形参列表和返回值，函数也能对数据进行加工，用全局变量即可。总的来说，函数参数传参用的比较多，因为这样可以实现模块化编程，而C语言中也是尽量减少使用全局变量。

(5)全局变量传参最大的好处就是省略了函数传参的开销，所以效率更高一点，但是实战中用的最多的还是传参，如果参数很多传参开销非常大，通常的做法是把很多参数打包成一个结构体，然后传结构体变量指针进去。

### 3.10.2.函数传参中使用const指针

(1)const一般用在函数参数列表中，用法是const int \*p;（意义是指针变量p本身是可变的，而p所指向的变量是不可变的）

(2)const用来修饰指针做函数传参，作用就在于声明在函数内部不会改变这个指针指向的内容，所以给该函数传一个不可改变的指针（char \*p = "linux";这种）不会触发错误，而一个未声明为const的指针的函数，你给他传一个不可更改的指针的时候就要小心了。

void func1(char \*p)

{

\*p = 'a';

}

void func2(const char \*p) // error,此函数在编译时就会因函数内部对const型变量赋值而编译错误

{

\*p = 'a';

}

int main()

{

char \*pStr = "linux"; // 指针方式表示的字符串存在于内存的代码段，是不可更改的

//char pStr[] = "linux"; // ok的,局部变量数组存在于内存的栈中

func1(pStr); // error,指针方式引用的字符串存在于内存的代码段，是不可更改的,这里使用它会造成段错误，指针指向了不该指向的地方

//func2(pStr); // error,此函数在编译时就会因函数内部对const型变量赋值而编译错误

printf("%s.\n", pStr);

return 0;

}

### 3.10.3.函数需要向外部返回多个值怎么办？

(1)一般来说，函数的输入部分就是函数参数，输出部分就是返回值。问题是函数的参数可以有很多个，而返回值只能有1个。这就造成我们无法一个函数返回多个值。

(2)实际编程中，一个函数需要返回多个值是非常普遍的，因此完全依赖于返回值是不靠谱的，通常的做法是用参数来做返回。（在典型linux风格函数中，返回值是不用来返回结果的，而是用来返回0或者负数来表示查询执行结果是对还是错，是成功还是失败，它是返回给调用它的进程的）。

(3)普遍做法，编程中函数的输入和输出都是靠函数参数的，返回值只是用来表示函数执行的结果是对还是错。如果这个参数是用来做输入的，就叫输入型参数；如果这个参数的目的是用来做输出的，就叫输出型参数。

(4)输出型参数就是用来让函数内部把数据输出到函数外部的。

例：传变量a进去，当a大于100时，返回-1，当a小于等于100时，执行a\*5，将结果赋值给输出型参数\*p，返回0。

int multip5(int a, int \*p) {

int tmp;

tmp = 5 \* a;

if (tmp > 100)

{

return -1;

}

else

{

\*p = tmp;

return 0;

}

}

int main(void)

{

int a, b = 0, ret = -1;

a = 30;

ret = multip5(a, &b);

if (ret == -1)

{

printf("出错了\n");

}

else

{

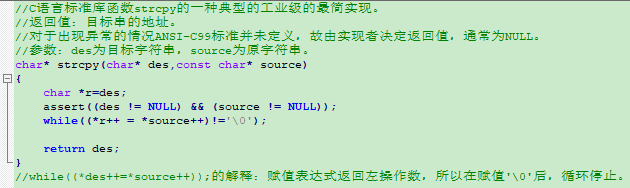
printf("result = %d.\n", b);

}

}

总结：

看到一个函数的原型后，怎么样一眼看出哪个参数做输入哪个参数做输出？函数传参如果传的是普通变量（不是指针）那肯定是输入型参数；如果穿指针就有2种可能性了，为了区别，经常的做法是：如果这个参数是做输入的（通常做输入的在函数内部只需要读取这个参数而不会需要更改它）就在指针前面加const来修饰；如果函数参数是指针变量并且还没加const，那么就表示这个参数是用来做输出型参数的。譬如C库函数中的strcpy函数。char \*strcpy(char\* dest, const char \*src);



# 四、C语言复杂表达式与指针高级应用

## 4.1指针数组与数组指针

### 4.1.1.字面意思来理解指针数组和数组指针

(1)指针数组的实质是数组，这个数组中存储的内容全部都是指针变量。

(2)数组指针的实质是指针，这个指针指向的是一个数组。

### 4.1.2.分析指针数组与数组指针表达式

(1)int \*p[5]; int (\*p)[5]; int \*(p[]);

(2)一般规律：int \*p;（p是一个指针）；int p[5]; （p是一个数组）

总结：我们在定义一个符号时，关键在于：首先要搞清楚你定义的符号是谁（第一步：找核心）；其次再来看谁跟核心最近、谁跟核心结合（第二部：找结合）；以后继续向外扩展（第三部：继续向外结合直到整个符号完）。

(3)如果核心和\*结合，表示核心是指针；如果核心和[]结合，表示核心是数组；如果核心和()结合，表示核心是函数。

(4)用一般规律分析三个符号：

第一个：int \*p[5];

核心是p，p是一个数组，数组中有5个元素，数组中的元素都是指针，指针指向的元素是int类型的变量，整个符号是一个指针数组。

第二个：int(\*p)[5];

核心是p，p是一个指针，指针指向一个数组，数组中有5个元素，数组中存的元素是int类型的变量，整个符号是一个数组指针。

第三个：int \*(p[5]);

解析方法和结论和第一个相同，()在这里是可有可无的。

注意：符号的优先级到底有什么用？是决定2个符号一起作用的时候决定哪个符号先运算，哪个符号后运算。

遇到优先级问题怎么办？第一，查优先级表；第二，自己记住（全部记住都成神了，人只要记住[] . ->这几个优先级比较高即可）。

总结

(1)优先级和结合性是分析符号意义的关键

在分析C语言问题时不要胡乱去猜测规律，不要总觉得C语言无从捉摸，从已知的规律出发按照既定的规则去做即可。

(2)学会逐层剥离的分析方法

找到核心后从内到外逐层的进行结合，结合之后可以把已经结合的部分当成一个整体，再去和整体外面的继续进行结合。

(3)基础理论和原则是关键，没有无缘无故的规则。

## 4.2 函数指针与typedef

### 4.2.1.函数指针的实质（还是指针变量）

(1)函数指针的实质还是指针，还是指针变量。本身占4字节（在32位系统中，所有的指针都是4字节）

(2)函数指针、结构体指针、数组指针、普通指针之间并没有本质区别，区别在于指针指向的东西是个什么玩意儿。

(3)函数的实质是一段代码，这一段代码在内存中是连续分布的（一个函数的大括号括起来的所有语句将来编译出来生成的可执行程序是连续的），所以对于函数来说很关键的就是函数中的第一句代码的地址，这个地址就是所谓的函数地址，在C语言中用函数名这个符号来表示。

(4)结合函数的实质，函数指针其实就是一个普通变量，这个普通变量的类型是函数指针变量类型，它的值就是某个函数的地址（也就是它的函数名这个符号在编译器中对应的值）。

### 4.2.2.函数指针的书写和分析方法

(1)C语言本身是强类型语言（每一个变量都有自己的变量类型），编译器可以帮我们做严格的类型检查。

int \*p;

int a[5];

p = a; // 类型匹配，所以编译器不会警告不会报错。

//p = &a; // 类型不匹配，p是int \*, &a是int (\*)[5];

int (\*p1)[5] ;

p1 = &a; // 类型匹配，p1类型是int (\*)[5]，&a的类型也是int (\*)[5]

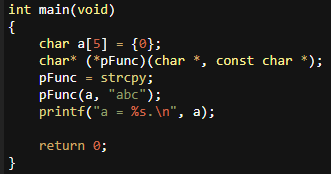
(2)所有的指针变量类型其实本质都是是一样的，但是为什么在C语言中要去区分它们，写法不一样呢（譬如int类型指针就写作int \*p; 数组指针就写作int (\*p)[5]; 函数指针就写的更复杂）

(3)假设我们有个函数是：void func(void);对应的函数指针就是void \*pFunc(void);类型就是void (\*)(void);

(4)函数名和数组名最大的区别是：函数名做右值时加不加&效果和意义都是一样的，但是数组名做右值时加不加&意义就不一样（加&表示整个数组的首地址；不加&表示数组的首元素地址，相当于&a[0]）。



(5)写一个复杂的函数指针的实例：譬如函数是strcpy函数：char \*strcpy(char \*dest, const char \*src);，对应的函数指针就是：char \*(\*pFunc)(char \*dest, const char \*src);



### 4.2.3.typedef关键字的用法

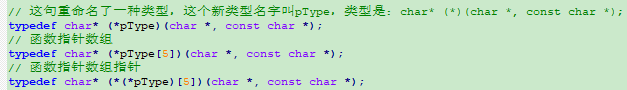
(1)typedef是C语言中的一个关键字，作用是用来定义（或者叫重命名类型）

(2)C语言中的类型一共有2种：一种是编译器定义的原生类型（基础数据类型，如int、double之类的）；第二种是用户自定义类型，不是语言自带的，是程序员自己定义的（譬如数组类型、结构体类型、函数类型......）。

(3)数组指针、指针数组、函数指针等都属于用户自定义类型。

(4)有时候自定义类型太长了，用起来不方便，所以用typedef给它重命名一个短点的名字。

(5)注意：typedef是给类型重命名，也就是说typedef加工出来的都是类型，而不是变量。



总结：函数指针的分析方法也是源于优先级与逐层剥离的基本理论。

## 4.3 函数指针实战

### 4.3.1.用函数指针调用执行函数

(1)用简单的函数指针来调用函数的示例，在上节课中已经演示过了。

(2)本节演示的是用函数指针指向不同的函数来实现同一个调用执行不同的结果。

(3)C++、C#、Java等面向对象的语言中有三大特征，其中有个多态。多态就是同一个执行实际结果不一样，跟我们这里看到的现象其实是一样的。

(4)刚才的调试过程，可以得到很多信息：

第一：当程序出现段错误时，第一步先定位段错误。定位的方法就是在可疑处加打印信息，从而锁定导致段错误的语句，然后集中分析这句为什么会段错误。

第二：linux中命令行默认是行缓冲的，意思就是说当我们程序printf输出的时候，linux不会一个字一个字的输出我们的内容，而是将其缓冲起来放在缓冲区等一行准备完了再一次性把一行全部输出出来（为了效率）。linux判断一行有没有完的依据就是换行符'\n'（windows中换行符是\r\n, linux中是\n，iOS中是\r）。也就是说你printf再多，只要没有遇到\n（或者程序终止，或者缓冲区满）都不会输出而会不断缓冲，这时候你是看不到内容输出的。因此，在每个printf打印语句（尤其是用来做调试的printf语句）后面一定要加\n，否则可能导致误判。

第三：关于在linux命令行下用scanf写交互性代码的问题，想说以下几点：

①命令行下的交互程序纯粹是用来学习编程用的，几乎没有实践意义，大家别浪费时间了。

②scanf是和系统的标准输入打交道，printf和标准输出打交道。要完全搞清楚这些东西得把标准输入标准输出搞清楚。

③我们用户在输入内容时结尾都会以\n结尾，但是程序中scanf的时候都不会去接收最后的\n，导致这个回车符还存留在标准输入中。下次再scanf时就会先被拿出来，这就导致你真正想拿的那个数反而没机会拿，导致错误。

#include <stdio.h>

int add(int a, int b);

int sub(int a, int b);

int multiply(int a, int b);

int divide(int a, int b);

// 定义了一个类型pFunc，这个函数指针类型指向一种特定参数列表和返回值的函数

typedef int (\*pFunc)(int, int);

int main(void)

{

pFunc p1 = NULL;

char c = 0;

int a = 0, b = 0, result = 0;

printf("请输入要操作的2个整数:\n");

scanf("%d %d", &a, &b);

printf("请输入操作类型：+ | - | \* | /\n");

do

{

scanf("%c", &c);

}while (c == '\n');

// 加一句调试

//printf("a = %d, b = %d, c = %d.\n", a, b, c);

switch (c)

{

case '+':

p1 = add; break;

case '-':

p1 = sub; break;

case '\*':

p1 = multiply; break;

case '/':

p1 = divide; break;

default:

p1 = NULL; break;

}

result = p1(a, b);

printf("%d %c %d = %d.\n", a, c, b, result);

return 0;

}

int add(int a, int b)

{

return a + b;

}

int sub(int a, int b)

{

return a - b;

}

int multiply(int a, int b)

{

return a \* b;

}

int divide(int a, int b)

{

return a / b;

}

### 4.3.2.结构体内嵌函数指针实现分层

(1)程序为什么要分层？因为复杂程序东西太多一个人搞不定，需要更多人协同工作，于是乎就要分工。要分工先分层，分层之后各个层次由不同的人完成，然后再彼此调用组合共同工作。

(2)本程序要完成一个计算器，我们设计了2个层次：上层是framework.c，实现应用程序框架；下层是cal.c，实现计算器。实际工作时cal.c是直接完成工作的，但是cal.c中的关键部分是调用的framework.c中的函数来完成的。

(3)先写framework.c，由一个人来完成。这个人在framework.c中需要完成计算器的业务逻辑，并且把相应的接口写在对应的头文件中发出来，将来别的层次的人用这个头文件来协同工作。

(4)另一个人来完成cal.c，实现具体的计算器；这个人需要framework层的工作人员提供头文件来工作（但是不需要framework.c）

(5)总结：

第一：本节和上节实际完成的是同一个习题，但是采用了不同的程序架构。

第二：对于简单问题来说，上节的不分层反而容易理解，反而简单；本节的分层代码不好理解，看起来有点把简单问题复杂化的意思。原因在于我们这个问题本身确实是简单问题，而简单问题就应该用简单方法处理。我们为什么明知错误还要这样做？目的是向大家演示这种分层的写代码的思路和方法。

第三：分层写代码的思路是：有多个层次结合来完成任务，每个层次专注各自不同的领域和任务；不同层次之间用头文件来交互。

第四：分层之后上层为下层提供服务，上层写的代码是为了在下层中被调用。

第五：上层注重业务逻辑，与我们最终的目标相直接关联，而没有具体干活的函数。

第六：下层注重实际干活的函数，注重为上层填充变量，并且将变量传递给上层中的函数（其实就是调用上层提供的接口函数）来完成任务。

第七：下层代码中其实核心是一个结构体变量（譬如本例中的struct cal\_t），写下层代码的逻辑其实很简单：第一步先定义结构体变量；第二步填充结构体变量；第三步调用上层写好的接口函数，把结构体变量传给它既可。