# 类型转换

## 隐式类型转换

整型、实型和字符型数据之间可以混合运算。例如：

10 + ’a’ + 1.5 - 8765.1234 \* ’b’

不同数据类型之间运算会进行自动类型转换

|  |
| --- |
| C:\Users\wudunxiong\AppData\Roaming\feiq\RichOle\1556803666.bmp |

## 强制类型转换

* 一般形式：（类型名）（表达式）

例 (int)(x + y)

(int)x + y

(double)(3/2)

(int)3.6

* 说明：**强制转换得到所需类型的中间变量，原变量类型、变量值保持不变**
* **较高类型向较低类型转换时可能发生精度损失问题**

|  |
| --- |
| **#include <stdio.h>**  **main()**  **{**  **float x;**  **int i;**  **x=3.6;**  **i=(int)x;**  **printf(“x=%f,i=%d”,x,i);**  **}**  **结果：x=3.600000,i=3** |

## 类型间转换

|  |
| --- |
| 例如  int a = 2147483648;  printf("%d",a);  这样赋值后，输出变量a的值并非预期的2147483648，而是-2147483648，原因是2147483648超出了int类型能够装载最大值，数据产生了溢出。如果换一种输出格式控制符，代码如下所示：  printf("%u",a);  输出的结果就是变量a的值，原因是%u是按照无符号整型输出的数据，而无符号整型的数据范围上限大于2147483648这个值。 |

|  |
| --- |
| * 当把占字节较小的数据赋值给占字节较大的数据时，可能出现以下两种情况。 * 第1种情况，当字节较大数是无符号数时，转换时新扩充的位被填充成0   char b = 10;  unsigned short a = b;  printf("%u",a);  这样赋值后，变量a中输出的值是10 |

|  |
| --- |
| * 当字节较大数是有符号数时，转换时新扩充的位被填充成符号位   char b = 255;  short a = b;  printf("%d",a);  这样赋值后，变量a输出的值是-1，变量a扩充的高8位，根据变量b的最高位1都被填充成了1，所以数值由正数变成了负数，因为变量a的最高位符号位是1，至于为什么16个1表示的是-1，涉及到二进制数的原码和补码问题 |

## 操作符分类

|  |
| --- |
|  |

## 运算符优先级

|  |
| --- |
|  |

# 数据类型

## 初学者数组类型三大难题

问题引出

1. 数组类型
2. 数组指针
3. 数组类型和数组指针的关系

## 数据类型的本质

从编译器和内存的角度理解数据类型：

Int a 告诉C编译器给我分配四个字节的内存。

1、数据类型的本质：可理解为创建变量的模具，是固定大小内存的别名。

2、数据类型的作用：

3、求数据类型大小：sizeof()

注意：sizeof()是操作符（单目运算符），不是函数。Sizeof测量的实体大小在编译期间就已经确定

4、数据类型可以有别名

typedef uu int；

sizeof（uu）=sizeof（int）；

## sizeof()

### sizeof的含义：

sizeof是C语言的一种单目操作符，如C语言的其他操作符++、--等。它并不是函数。sizeof操作符以字节形式给出了其操作数的存储大小。操作数可以是一个表达式或括在括号内的类型名。操作数的存储大小由操作数的类型决定。

### **Sizeof的用法**：

1. 用于数据类型

数据类型必须用括号括住。如sizeof（int）

1. 用于变量

sizeof使用形式：sizeof（var\_name）或sizeof var\_name

　注意：sizeof操作符不能用**于函数类型，不完全类型或位字段**。不完全类型指具有未知存储大小的数据类型，如未知存储大小的数组类型、未知内容的结构或联合类型、void类型

3、sizeof的结果   
　　sizeof操作符的结果类型是size\_t，它在头文件<stddef.h>中typedef为unsigned int类型。该类型保证能容纳实现所建立的最大对象的字节大小。   
  
　　1、若操作数具有类型char、unsigned char或signed char，其结果等于1。   
　　ANSI C正式规定字符类型为1字节。   
  
　　2、int、unsigned int 、short int、unsigned short 、long int 、unsigned long 、float、double、long double类型的sizeof 在ANSI C中没有具体规定，大小依赖于实现，一般可能分别为2、2、2、2、4、4、4、8、10。   
  
　　3、当操作数是指针时，sizeof依赖于编译器。例如Microsoft C/C++7.0中，near类指针字节数为2，far、huge类指针字节数为4。一般Unix的指针字节数为4。   
  
　　4、当操作数具有数组类型时，其结果是数组的总字节数。   
  
　　5、联合类型操作数的sizeof是其最大字节成员的字节数。结构类型操作数的sizeof是这种类型对象的总字节数，包括任何垫补在内。   
  
　　让我们看如下结构：   
  
　　struct {char b; double x;} a;   
  
　　在某些机器上sizeof（a）=12，而一般sizeof（char）+ sizeof（double）=9。   
  
　　这是因为编译器在考虑对齐问题时，在结构中插入空位以控制各成员对象的地址对齐。如double类型的结构成员x要放在被4整除的地址。   
  
　　6、如果操作数是函数中的数组形参或函数类型的形参，sizeof给出其指针的大小

## void类型

### void的含义：

void的出现只是为了一种抽象的需要，如果你正确地理解了面向对象中“抽象基类”的概念，也很容易理解void数据类型。正如不能给抽象基类定义一个实例，我们也不能定义一个void（让我们类比的称void为“抽象数据类型”）变量。

Void的字面意思是“无类型”，void\*类型是“无类型指针”，void \*可以指向任何类型的数据。void几乎只有“注释”和限制程序的作用，定义一个void变量没有意义。

void例如：

void a;

这行语句编译时会出错，提示“illegal use of type 'void'”。不过，即使void a的编译不会出错，它也没有任何实际意义。

void真正发挥的作用在于：

　　（1） 对函数返回的限定；

　　（2） 对函数参数的限定。

众所周知，如果指针p1和p2的类型相同，那么我们可以直接在p1和p2间互相赋值；如果p1和p2指向不同的数据类型，则必须使用强制类型转换运算符把赋值运算符右边的指针类型转换为左边指针的类型。

void\*例如：

　　float \*p1;

　　int \*p2;

　　p1 = p2;

其中p1 = p2语句会编译出错，提示“'=' : cannot convert from 'int \*' to 'float \*'”，必须改为：

p1 = (float \*)p2;

而void \*则不同，任何类型的指针都可以直接赋值给它，无需进行强制类型转换：

　　void \*p1;

　　int \*p2;

　　p1 = p2;

　　但这并不意味着，void \*也可以无需强制类型转换地赋给其它类型的指针。因为“无类型”可以包容“有类型”，而“有类型”则不能包容“无类型”。道理很简单，我们可以说“男人和女人都是人”，但不能说“人是男人”或者“人是女人”。下面的语句编译出错：

　　void \*p1;

　　int \*p2;

　　p2 = p1;

　　提示“'=' : cannot convert from 'void \*' to 'int \*'”。

### Void与void\*的实用规则：

**规则一：如果函数没有返回值，那么应声明为void类型**

　　在C语言中，凡不加返回值类型限定的函数，就会被编译器作为返回整型值处理。但是许多程序员却误以为其为void类型。例如：

　　add ( int a, int b )

　　{

　　return a + b;

　　}

　　int main(int argc, char\* argv[])

　　{

　　printf ( "2 + 3 = %d", add ( 2, 3) );

　　}

　　程序运行的结果为输出：

　　2 + 3 = 5

　　这说明不加返回值说明的函数的确为int函数。

[林锐](http://baike.baidu.com/view/962056.htm)博士《高质量C/C++编程》中提到：“C++语言有很严格的类型安全检查，不允许上述情况（指函数不加类型声明）发生”。可是编译器并不一定这么认定，譬如在Visual C++6.0中上述add函数的编译无错也无警告且运行正确，所以不能寄希望于编译器会做严格的类型检查。

　　因此，为了避免混乱，在编写C/C++程序时，对于任何函数都必须一个不漏地指定其类型。如果函数没有返回值，一定要声明为void类型。这既是程序良好可读性的需要，也是编程规范性的要求。另外，加上void类型声明后，也可以发挥代码的“自注释”作用。代码的“自注释”即代码能自己注释自己。

**规则二：如果函数无参数,那么应声明其参数为void。**

　　在C++语言中声明一个这样的函数：

　　int function(void)

　　{

　　return 1;

　　}

　　则进行下面的调用是不合法的：

　　function(2);

　　因为在C++中，函数参数为void的意思是这个函数不接受任何参数。

　　在Turbo C 2.0中编译：

　　#include "[stdio.h](http://baike.baidu.com/view/538727.htm)"

　　fun()

　　{

　　return 1;

　　}

　　main()

　　{

　　printf("%d",fun(2));

　　getchar();

　　}

　　编译正确且输出1，这说明，在C语言中，可以给无参数的函数传送任意类型的参数，但是在C++编译器中编译同样的代码则会出错。在C++中，不能向无参数的函数传送任何参数，出错提示“'fun' : function does not take 1 parameters”。

　　所以，无论在C还是C++中，若函数不接受任何参数，一定要指明参数为void。

**规则三：小心使用void指针类型**

　　按照ANSI(American National Standards Institute)标准，不能对void指针进行算法操作，即下列操作都是不合法的：

　　void \* pvoid;

　　pvoid++; //ANSI：错误

　　pvoid += 1; //ANSI：错误

　　//ANSI标准之所以这样认定，是因为它坚持：进行算法操作的指针必须是确定知道其指向数据类型大小的。

　　//例如：

　　int \*pint;

　　pint++; //ANSI：正确

　　pint++的结果是使其增大sizeof(int)。( 在VC6.0上测试是sizeof(int)的倍数)

　　但是大名鼎鼎的GNU(GNU's Not Unix的缩写)则不这么认定，它指定void \*的算法操作与char \*一致。

　　因此下列语句在GNU编译器中皆正确：

　　pvoid++; //GNU：正确

　　pvoid += 1; //GNU：正确

　　pvoid++的执行结果是其增大了1。( 在VC6.0上测试是sizeof(int)的倍数)

　　在实际的程序设计中，为迎合ANSI标准，并提高程序的可移植性，我们可以这样编写实现同样功能的代码：

　　void \* pvoid;

　　(char \*)pvoid++; //ANSI：正确；GNU：正确

　　(char \*)pvoid += 1; //ANSI：错误；GNU：正确

　　GNU和ANSI还有一些区别，总体而言，GNU较ANSI更“开放”，提供了对更多语法的支持。但是我们在真实设计时，还是应该尽可能地迎合ANSI标准。

**规则四：用以数据类型的封装，作函数参数和返回值**

如果函数的参数可以是任意类型指针，那么应声明其参数为void \*

　　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 );

　　这样，任何类型的指针都可以传入memcpy和memset中，这也真实地体现了内存操作函数的意义，因为它操作的对象仅仅是一片内存，而不论这片内存是什么类型。如果memcpy和memset的参数类型不是void \*，而是char \*，那才叫真的奇怪了！这样的memcpy和memset明显不是一个“纯粹的，脱离低级趣味的”函数！

　　下面的代码执行正确：

　　//示例：memset接受任意类型指针

　　int intarray[100];

　　memset ( intarray, 0, 100\*sizeof(int) ); //将intarray清0

　　//示例：memcpy接受任意类型指针

　　int intarray1[100], intarray2[100];

　　memcpy ( intarray1, intarray2, 100\*sizeof(int) ); //将intarray2拷贝给intarray1

　　有趣的是，memcpy和memset函数返回的也是void \*类型，标准库函数的编写者是多么地富有学问啊！

**规则五：void不能代表一个真实的变量**

　　下面代码都企图让void代表一个真实的变量，因此都是错误的代码：

　　void a; //错误

　　function(void a); //错误

　　void体现了一种抽象，这个世界上的变量都是“有类型”的，譬如一个人不是男人就是女人（还有人妖？）。

# 变量

## 变量的概念及本质：

**变量命名规则：**

标示符：

1、程序中用于标识常量、变量、函数的字符序列。

2、只能由字母、数字、下划线组成

3、第一个字母必须是字母或下划线，大小写有区别，不能使用C语言的关键字。

4、如果变量不初始化，就会默认读取垃圾数据， 有些垃圾数据会导致程序崩溃。

**变量的本质**：一段连续内存空间的别名

1. 程序通过变量来申请和命名内存空间 int a = 0;（变量名a在代码区，是0这个四字节内存空间的别名）
2. 通过变量访问内存空间

## 修改变量的三种方法

1. 直接修改：int a = 10; a=20;
2. 间接修改：内存有地址编号，拿到地址编号也可以修改内存；

外挂原理就是通过变量内存地址修改变量值

&a=1245024;

\*((int \*)(1245024)) = 10;

1. C++中引用

## 数据类型和变量的关系

C语言规定：通过数据类型来定义一个变量

## 声明变量的意义

C语言为什么要规定先声明变量呢？为什么要指定变量的名字和对应的数据类型呢？

（1）建立变量符号表。

通过声明变量，编译器可以建立变量符号表，如此一来，程序中用到了多少变量，每个变量的类型是什么，编译器非常清楚，是否使用了没有声明的变量，编译器在编译期间就可以发现。从而帮助了程序员远离由于疏忽而将变量名写错的情况。

（2）变量的数据类型指示系统分配多少内存空间。

（3）变量的数据类型指示了系统如何解释存储空间中的值。

同样的数值，不同的类型将有不同的解释。int占据4个字节，float也占据4个字节，在内存中同样也是存储的二进制数，并且这个二进制数也没有标志区分当前是int型还是float型。如何区分？就是通过变量的数据类型来区分。由于声明建立了变量符号表，所以系统知道变量该如何解释。

（4）变量的数据类型确定了该变量的取值范围

例如短整型数据取值-32767～32767之间。

（5）不同的数据类型有不同的操作

如整数可以求余。C语言用符号”%”表示求余。整数可以，实数不可

# 常量

# 二：函数调用模型

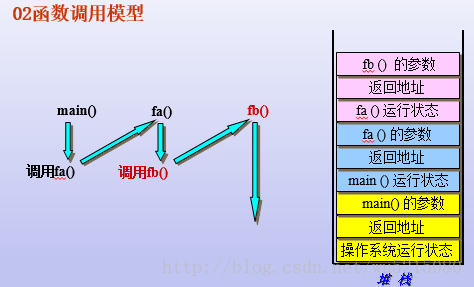
变量三要素是：名称、大小、作用域。那么变量的生命周期是多长呢？

编译器是如何管理每个函数间变量的生命周期呢？

要研究变量的生命周期，而变量一般又是在函数中定义分配空间的。

因此下面研究一下变量作为函数参数和返回值传递分析

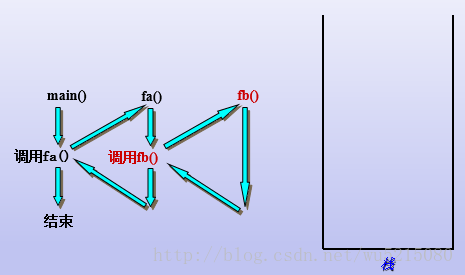
下面我们具体总结一下，各个函数的变量的生命周期



main里面的变量分配内存，函数fa(),函数fb()中的变量分配的内存空间它们的生命周期都是多长呢？

上述图1，已经说明了内存主要分为四区，因此每个函数中变量在堆栈的生命周期是不同的，

同时在函数调用的时候，先执行的函数最后才执行完毕



char\*fa()  
{

char\*pa = "123456";//pa指针在栈区，“123456”在常量区，该函数调用完后指针变量pa就被释放了  
 char\*p = NULL;     //指针变量p在栈中分配4字节  
 p=(char\*)malloc(100);//本函数在这里开辟了一块堆区的内存空间，并把地址赋值给p  
 strcpy(p, "wudunxiong 1234566");//把常量区的字符串拷贝到堆区  
 return p;//返回给主调函数fb()，相对fa来说fb是主调函数，相对main来说，fa(),fb()都是被调用函数  
}  
char\*fb()  
{  
 char\*pstr = NULL;  
 pstr = fa();  
 return pstr;//指针变量pstr在这就结束  
}

void main()  
{   
 char\*str = NULL;  
 str = fb();  
 printf("str = %s\n",str);  
 free(str);    //防止内存泄露，被调函数fa()分配的内存存的值通过返回值传给主调函数，然后主调函数释放内存  
 str = NULL;//防止产生野指针  
 system("pause");  
}

总结：

1、主调函数分配的内存空间（堆，栈，全局区）可以在被调用函数中使用，可以以指针作函数参数的形式来使用

2、被调用函数分配的内存空间只有堆区和全局区可以在主调函数中使用（返回值和函数参数），而栈区却不行，因为栈区函数体运行完之后

这个函数占用的内存编译器自动帮你释放了。

3、一定要明白函数的主被调关系以及主被调函数内存分配回收，也就是后面接下几篇总结的函数的输入输出内存模型

# 内存四区模型

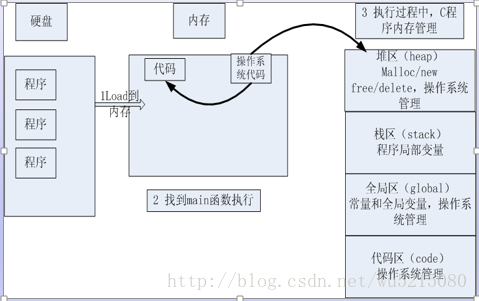


                           图1、内存四区模型  
流程说明  
1、操作系统把物理硬盘代码load到内存  
2、操作系统把c代码分成四个区  
3、操作系统找到main函数入口执行

## 1、内存四区：

一个由c/C++编译的程序占用的内存分为以下几个部分  
1、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。其操作方

式类似于数据结构中的栈。  
2、堆区（heap： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回   收 。

注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3、数据区：主要包括静态全局区和常量区，如果要站在汇编角度细分的话还可以分为很多小的区。  
           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。

程序结束后有系统释放  
常量区 ：常量字符串就是放在这里的。 程序结束后由系统释放

4、代码区：存放函数体的二进制代码。

## 2、堆栈的生长方向：

  1、堆栈的生长方向和存放数据增长方向不一样

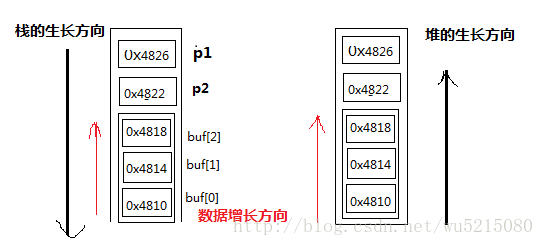


                         图2、堆栈的生长方向

# 指针专题一：

## 一级指针易错模型：

1、不断修改指针的指向，然后又free和printf

2、Char buf[3] = “asc;”

3、\*p++与（\*p）++.；前者是指针++,后者是数据++

一遍可能定义一个变量计数值的时候用\*count++，错误用法

还有就是求字符串的时候指针指向改变，然后释放该内存的时候出错

1. 还有就是被调用函数分配临时空间，然后把该临时空间首地址返回出来出错
2. 解决的方法有多种，一般字符串指针的时候用两个指针，一个用于记录首地址
3. 为什么int a[10];中a是个常量

a++;//这里会报错，为什么数组首地址不能++

因为a是数组类型的内存块的首地址，这数组是个临时变量。编译器要拿着首地址a去释放内存，为了避免你把a 的指向改变所以设置为常量

# 指针专题二：指针做函参间接赋值是指针存在最大意义

## 1、\*p间接赋值成立条件：3个条件

1)2个变量（通常一个实参，一个形参）

2) 建立关系，实参取地址赋给形参指针

3）\*p形参去间接修改实参的值

## 2、间接赋值的应用场景：三个条件组合

1）：123都写在一个函数里面

2）：12写在一个函数里面 3 写在另外一个函数里面

3）：1 写在一个函数里面 23 写在另外一个函数里面（C++中引用）

## 3、引申： 函数调用时,用n指针（形参）改变n-1指针（实参）的值。

用1级指针形参，去间接修改了0级指针（实参）的值。。

用2级指针形参，去间接修改了1级指针（实参）的值。。

用3级指针形参，去间接修改了2级指针（实参）的值。。

用n级指针形参，去间接修改了n-1级指针（实参）的值。。

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。这是指针作为函数参数的精髓。

# 指针专题三：指针作函数参数输入输出特性

1）、理解指针要把内存四区模型和函数调用模型相结合

2）、主调，被调函数谁分配内存

主调用函数分配内存，被调用函数使用内存---输入

被调用函数分配内存把结果输出给主调用内存主要通过（指针做函数参数和返回值做输出），主调用函数析构该内存

3）、指针作函数参数的输入输出特性

# 数组专题

## 数组初始化：

数组元素的个数可以显示或隐式指定

|  |
| --- |
| int main()  {  int a[10] = {1,2};//其他没初始化元素，编译器默认帮你初始化为0  int b[] = {1, 2};//编译器隐式指定长度为两个元素  int c[20] = {0};  for (i=0; i<10; i++)  {  printf("%d ", a[i]);  }  memset(a, 0, sizeof(a));  getchar();  } |

## 2、数组名理解难点

|  |
| --- |
| int a[10]={1,2};  printf("&a:%d,a:%d\n",&a,a);  printf("&a+1:%d,a+1:%d\n", &a+1, a+1); |

1. 数组首元素地址与数组地址：数组名a代表元素首地址,&a代表数组地址。
2. 数组首元素地址=数组地址即：a=&a
3. &a+1代表的是加一个数组长度sizeof(a)，a+1代表的是加一个数组元素长度sizeof(a[0])
4. 一维数组名是一个指针常量
5. **二维数组名是一个数组指针,多维数组名也是一个数组指针,是指向一个低维数组的指针**

|  |
| --- |
| char array[10][30];  (array+i) //相当于 第i行的首地址 //二级指针  （\*(array+i)） //一维数组的首地址 //一级指针  \*(array+i)+j //相当于第i行第j列的地址 &array[i][j]  \*(\*(array+i)+j) //相当于第i行第j列的地址 array[i][j] |

6、多维数组在物理内存中是线性存储的，只是编译器帮我们优化了

|  |
| --- |
| C:\Users\wudunxiong\AppData\Roaming\Tencent\Users\285117138\QQ\WinTemp\RichOle\SAHIF%G7$KBKUS1)N)G%TJY.jpg |

## 2、初学者数组类型三大难题

数组类型

数组指针：数组是一个指针，该指针是指向一个数组的指针（数组类型和数组指针的关系）

指针数组：指针是数组元素，也就是数组元素中存储的都是指针

## 数组类型

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**

1、定义一个数组类型 ：用数组定义变量

|  |
| --- |
| void main12()  {  int i = 0;  //定义一个数组类型  typedef int MyArrayType[5]; //int  MyArrayType myArray; //int myArray[5];  for (i=0; i<5; i++)  {  myArray[i] = i +1;  }  for (i=0; i<5; i++)  {  printf("%d \n", myArray[i]);  }  system("pause");  } |

2、定义一个数组类型，用数组顶一个数组类型的指针

|  |
| --- |
| void main13()  {  int i = 0;  typedef int MyArrayType[5]; //定义一个数组类型  MyArrayType \*pArray = NULL; //定义了一个 数组类型的指针  Int a[5]={0};  pArray = &a;  for (i=0; i<5; i++) //我通过数组指针的方式来操作a[5]这块内存  {  (\*pArray)[i] = i+1; //a[i] = i+1;  }  for (i=0; i<5; i++)  {  printf("%d \n", (\*pArray)[i]);  }  system("pause");  } |

## 数组指针

定义数组指针

|  |
| --- |
| int i = 0;  //这个是定义了一个类型，这个类型是数组类型  typedef int MyArrayType[5]; //int  //这个是定义了一个类型，定义了一个指针类型，定义了一个指向数组的指针类型。。。。  typedef int (\*MyPArrType)[5] ; //数组指针  MyPArrType myPoint; //  int b[5];  myPoint = &b; //变量取地址给指针赋值  for (i=0; i<5; i++)  {  (\*myPoint)[i] = i+1;  }  for (i=0; i<5; i++)  {  printf("%d ", (\*myPoint)[i]);  }  system("pause");  } |
| 1）：typedef int MyArrayType[5]; //定义一个数组类型  MyArrayType \*pArray = NULL; //定义了一个 数组类型的指针  //这个是定义了一个类型，定义了一个指针类型，定义了一个指向数组的指针类型  2）：typedef int (\*MyPArrType)[5] ; //数组指针  MyPArrType myPoint; //int b[5];  3）：int (\*myArrayPoint)[5] ; //告诉编译给我开辟四个字节内存 |

## 5、指针数组

1、指针数组与数组指针容易混淆

|  |
| --- |
| Char \*p1[] = {“123”,”456”,”789”};  []优先级高，先与p结合成为一个数组，再由char\*说明这是一个字符型指针数组  //这是一个指针数组、数组3\*4、三个一维数组的首地址存放在以指针作为元素的数组中  Char (\*p2)[] = {“123”,”456”,”789”};  //编译器只分配4个字节，是一个指针，编译不通过、错误的方式 |

2、指针数组做函数参数退化

|  |
| --- |
| int printfArray(char \*buf [30]);  int printfArray(char \*buf[]);  int printfArray(char \*\*buf); |

1. 指针数组的两种用途

菜单

命令行

1. 指针数组自我结束的三种方法

|  |
| --- |
| char\* c\_keyword[] = { "while", "case","static","do",'\0'};  char\* c\_keyword[] = {"while", "case","static","do",0};  char\* c\_keyword[] = {"while", "case","static","do",NULL}; |

## 6、当多维数组当做函数参数的话的时候会退化为指针

|  |
| --- |
| **退化原因的本质是因为程序员眼中的二维内存，在物理内存上是线性存储**  //总结：函数调用的时候，把数组首地址和有效数据长度传给被调用函数才是最正确的做法 |
| **一维数组做函数参数退化过程**  //int a[10] -=-->int a[] ---->int \*a  //数组做函数形参的时候，如果在形参中定义int a[10]语句，  //c/c++编译器 会做优化，技术推演如下  //int a[10] -=-->int a[] ---->int \*a |
| **一维数组做函数参数退化过程**  char buf[10][30])—》char buf[][30])----》char (\*buf)[30]) |
| int printfArray(char buf[10][30]);  int printfArray(char buf[][30]);  int printfArray(char (\*buf)[30]);  这三者效果是一样的，也就验证了数组做函数参数退化为指针 |

第一种做法

|  |
| --- |
| int printfArray(int a[])  {  int i = 0;  printf("排序之前\n ");  for (i=0; i<10; i++)  {  printf("%d ", a[i]);  }  return 0;  } |
| 第二种写法 |
| int printfArray04(int \*a, int num)  {  int i = 0;  printf("排序之前\n ");  for (i=0; i<num; i++)  {  printf("%d ", a[i]);  }  return 0;  } |

|  |
| --- |
| 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&a[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] |

## 7、数组操作基础以及中括号本质

//c语言里面没有字符串这种类型。通过字符数组来模拟字符串

//C风格字符串是以零结尾的字符串

//操作数组的方法:**下标法和指针法**

|  |
| --- |
| void main()  {  int i = 0;  char \*p = NULL;  //通过字符串初始化字符数组 并且追加\0  char buf4[] = "abcd";  for (i=0; i<strlen(buf4); i++)  {  printf("%c", buf4[i]); //p[]  }  **//[] \*的本质到底是什么？**  **//\*p 是我们程序员手工的（显示）去利用间接赋值**  **//[] 只不过是，c/c++ 编译器帮我们做了一个\*p的操作。。。。。。**  **// buf4[i]======> buf4[0+i] ====> \*(buf4+i)**  **//===\*(buf4+i) --> bu4[i];**  printf("\n");  p = buf4;  for (i=0; i<strlen(buf4); i++)  {  printf("%c", \*(p+i)); //\*p  }  system("pause");  } |
| void main12()  {  **//字符数组初始化**  //指定长度 如果定义的长度剩余部分补充0  char buf1[100] = {'a', 'b', 'c'};  //不指定长度  char buf2[] = {'a', 'b', 'c'};  char buf3[] = {'a', 'b', 'c','\0'};  **//通过字符串初始化字符数组 并且追加\0**  char buf4[] = "abcd";  printf("%s\n", buf4 );  printf("sizeof(buf4): %d\n ", sizeof(buf4));  **//注意sizeof是对数据类型进行大小测量也就是数组类型 包括了\0**  printf("strlen(buf4): %d \n", strlen(buf4));**//strlen是求字符串的长度不包括\0**  system("pause");  } |

## 9：数组元素和数组地址的区别

数组元素Str[i] 与 str+i数组下标 strncpy（）函数中

# 指针专题四：二级指针做函数参数

## 1、二级指针第一种内存模型：常量区

|  |
| --- |
| int sortArray07(char\*\*pstr, int num)  {  int i = 0, j = 0;  int ret = 0;  char\*tmp = NULL;  if (pstr == NULL)  {  ret = -1;  printf("sortArray07() param err:%d\n", ret);  return ret;  }  for (i = 0; i < num; i++)  {  for (j = i + 1; j < num; j++)  {  if (strcmp(pstr[i], pstr[j])>0)  {  tmp = pstr[i];  pstr[i] = pstr[j];  pstr[j] = tmp;  }  }  }  return ret;  }  void main07()  {  char\*myArray[] = { "bbbb", "111111", "aaaaa", "ccccc" };  printf("排序前\n");  printArray07(myArray, 4);  sortArray07(myArray, 4);  printf("排序后\n");  printArray07(myArray, 4);  system("pause");  } |

## 2、二级指针第二种内存模型：栈区

|  |
| --- |
| int sortArray08(char (\*pstr)[10], int num)  {  int i = 0, j = 0;  char tmp[1024] = { 0 };  for (i = 0; i < num; i++)  {  for (j = i + 1; j < num; j++)  {  if (strcmp(pstr[i], pstr[j])>0)  {  strcpy(tmp, pstr[i]);  strcpy(pstr[i], pstr[j]);  strcpy(pstr[j], tmp);  }  }  }  return 0;  }  void main08()  {  char myArray[][10] = { "bbbb", "111111", "aaaaa", "ccccc" };  printf("二级指针第二种内存模型排序前\n");  printArray08(myArray, 4);  sortArray08(myArray, 4);  printf("二级指针第二种内存模型排序后\n");  printArray08(myArray, 4);  system("pause");  } |

## 3、二级指针第三种内存模型：堆区

|  |
| --- |
| //交换的是指针变量所指向空间的内容  int sortArray091(char\*\*pstr, int num)  {  int i = 0, j = 0;  char tmp[1024] = { 0 };  for (i = 0; i < num; i++)  {  for (j = i + 1; j < num; j++)  {  if (strcmp(pstr[i], pstr[j])>0)  {  strcpy(tmp ,pstr[i]);  strcpy(pstr[i], pstr[j]);  strcpy(pstr[j], tmp);  }  }  }  return 0;  }  //交换的是指针变量，排序  int sortArray092(char\*\*pstr, int num)  {  int i = 0, j = 0;  char\*tmp = NULL;  for (i = 0; i < num; i++)  {  for (j = i + 1; j < num; j++)  {  if (strcmp(pstr[i], pstr[j])>0)  {  tmp = pstr[i];  pstr[i] = pstr[j];  pstr[j] = tmp;  }  }  }  return 0;  }  char\*\*getMem( int num)  {  char\*\*myArray = (char\*\*)malloc(100 \* sizeof(char\*));  for (int i = 0; i < num; i++)  {  myArray[i] = (char\*)malloc(12 \* sizeof(char));  }  return myArray;  }  void main()  {  char\*\*myArray = NULL;  myArray = getMem(4);  if (myArray == NULL)  {  printf("malloc err\n");  return;  }  strcpy(myArray[0], "bbbb");  strcpy(myArray[1], "111111");  strcpy(myArray[2], "aaaaa");  strcpy(myArray[3], "ccccc");  printf("二级指针第三种内存模型排序前\n");  printArray09(myArray, 4);  //交换指针变量  //sortArray091(myArray, 4);  //交换指针变量所指向的空间  sortArray092(myArray, 4);  printf("二级指针第三种内存模型排序后\n");  printArray09(myArray, 4);  system("pause");  } |

# 野指针产生原因和解决办法

## 1）、产生问题分析：

|  |
| --- |
| void main22()  {  char \*p = NULL;  p = (char \*)malloc(100); //char p[100];  strcpy(p, "abcdefg");  if (p != NULL)  {  free(p); **//指针所指向的内存空间已经改变、但是指向没有改变**  }  if (p != NULL)//指针变量还是指向原来的内存空间，但是内存空间已经被释放  {  free(p);//因此这里再次释放程序就会bug  }  system("pause");  } |

**产生野指针的本质**：指针变量和它所指内存空间变量是两个不同的概念。

## 2）、解决办法：三步曲

1、定义指针时，把指针变量赋值成NULL

2、释放内存是，先判断指针变量是否为NULL

3、释放完内存后，把指针变量重新复制成NULL

## 3）、野指针产生模型图

|  |
| --- |
| C:\Users\wudunxiong\Desktop\临时文件夹\C\野指针2.png |

## 4）、野指针易错难点分析：指针做函数参数

|  |
| --- |
| void getMem3(int count, char \*p)  {  char \*tmp = NULL;  tmp = (char \*)malloc(100 \* sizeof(char)); //char tmp[100];  p = tmp;  //在这个场景下，你给形参赋值了，没有给实参赋值 ,因此分配空间的地址并没传给实参  }  void getMem4(int count, char \*\*p /\*out\*/)  {  char \*tmp = NULL;  tmp = (char \*)malloc(100 \* sizeof(char)); //char tmp[100];  //间接的修改实参  \*p = tmp;  }  //函数调用的时候，这个场景修改不了实参  int FreeMem1(char \*p)  {  if (p == NULL)  {  return -1;  }  if (p != NULL)  {  free(p);  p = NULL; //只是把形参重置为NULL，但是实参并没有改变  }  return 0;  }  int FreeMem2(char \*\*p) //实参地址传过来了  {  if (\*p == NULL)  {  printf("内存已经释放，不会产生野指针\n");  return -1;  }  if (\*p != NULL)  {  free(\*p);  \*p = NULL; //只是把实参重置为NULL，再也不会产生野指针了  }  return 0;  }  void main()  {  char \*myp = NULL;  //getMem3(100, myp);  getMem4(100, &myp);  strcpy(myp, "hello 123456");  FreeMem1(myp);  FreeMem2(&myp);  system("pause");  } |

**产生问题的本质原因**：指针做函数参数中形参和实参是两个不同的概念。没有真正理解指针做函数参数中地址传递和值传递的区别，这是典型的易错模型

# 结构体专题

## 02、点操作和指针操作本质研究

|  |
| --- |
| void main()  {  Teacher t1;  Teacher t2;  Teacher \*p = NULL;  printf(" %d \n", sizeof( Teacher));  p = &t1;  strcpy(t1.name, "name");  t1.age = 10; //通过 “.”的方法来操作结构体的成员域  p->age = 12;  p->age; // . ->的本质是寻址。。。。。寻每一个成员相对于大变量t1的内存偏移。。。。。。没有操作内存  //所以这样写是没有问题的。  t2 = t1; //编译器做了什么工作 |
|  |
|  |

## 03、编译器浅copy操作

|  |
| --- |
| 对结构体而言，指针做函数参数和元素变量做函数不同地方 |
| void copyStruct(Teacher \*to, Teacher \*from)  {  \*to = \*from;  }  //  int copyStruct2(Teacher to, Teacher from)  {  to = from;  return 10;  } |

## 04、结构体中套一级指针和二级指针 项目开发要点

|  |
| --- |
|  |
| Teacher \*creatTArray2(int num)  {  int i = 0, j = 0;  Teacher \*tArray = NULL;  tArray = (Teacher \*)malloc(num \* sizeof(Teacher));  if (tArray == NULL)  {  return NULL;  }  for (i=0; i<num; i++)  {  tArray[i].tile = (char \*)malloc(100);  }  //创建老师带的学生  for (i=0; i<num; i++)  {  char \*\*ptmp = (char \*\*)malloc((3+1)\*sizeof(char \*));  for (j=0; j<3; j++)  {  ptmp[j] = (char \*)malloc(120);  }  //ptmp[3] = NULL;  tArray[i].pStuArray = ptmp;  }  return tArray;  } |
| 释放函数 |
| int FreeTArray(Teacher \*tArray, int num)  {  int i =0, j = 0;  if (tArray == NULL)  {  return -1;  }  for (i=0; i<num; i++)  {  char \*\*tmp = tArray[i].pStuArray;  if (tmp ==NULL)  {  continue;;  }  for (j=0; j<3; j++)  {  if (tmp[j] != NULL)  {  free(tmp[j]);  }  }  free(tmp);  }  for (i=0; i<3; i++)  {  if (tArray[i].tile != NULL)  {  free(tArray[i].tile);  tArray[i].tile = NULL; //laji  }  }  free(tArray);  tArray = NULL; //垃圾  } |

## 05、深copy和浅copy

|  |
| --- |
| //产生的原因  //编译器给我们提供的copy行为是一个浅copy  //当结构体成员域中含有buf的时候，没有问题  //当结构体成员域中还有指针的时候，编译器只会进行指针变量的copy。指针变量所指的内存空间，编译器不会在多分分配内存  //这就是编译器的浅copy，我们要属顺从。。。。  // |
| /结构体的定义  typedef struct \_AdvTeacher  {  char \*name;  char buf[100];  int age;  }Teacher ;  Teacher \* creatT()  {  Teacher \*tmp = NULL;  tmp = (Teacher \*)malloc(sizeof(Teacher));  tmp->name = (char \*)malloc(100);  return tmp;  }  void FreeT(Teacher \*t)  {  if (t == NULL)  {  return ;  }  if (t->name != NULL)  {  free(t->name);  }  }  //解决方案  int copyObj(Teacher \*to, Teacher \*from)  {  //\*to = \*from;//copy；  memcpy(to, from, sizeof(Teacher));  to->name = (char \*)malloc(100);  strcpy(to->name, from->name);  } |
|  |

## 06、结体的高级话题

|  |
| --- |
| 深刻理解-》 。操作符的本质 |
| #include "stdlib.h"  #include "stdio.h"  #include "string.h"  typedef struct \_A  {  int a ;  };  //结构体的定义  typedef struct \_AdvTeacher  {  char \*name; //4  int age2 ;  char buf[32]; //32  int age; //4  struct \_A  }Teacher ;  void main2()  {  int i = 0;  Teacher \* p = NULL;  p = p - 1;  p = p - 2;  p = p +2;  p = p -p;  i = (int) (&(p->age)); //1逻辑计算在cpu中，运算  printf("i:%d \n", i);  //&属于cpu的计算，没有读写内存，所以说没有coredown  system("pause");  }  //-> .  void main()  {  int i = 0;  i = (int )&(((Teacher \*)0)->age );  printf("i:%d \n", i);  //&属于cpu的计算，没有读写内存，所以说没有coredown -->  system("pause");  } |

# 文件专题

**学文件读写一定要有自己的积累：小项目（简历）**

**将一级指针，二级指针内存模型，串联起来**

## 文件知识点总结：

Unsigned char修饰的表示不是以0结尾的字符

1、FILE文件结构体

|  |
| --- |
| struct \_iobuf {  char \*\_ptr; //文件输入的下一个位置  int \_cnt; //当前缓冲区的相对位置  char \*\_base; //指基础位置(应该是文件的其始位置)  int \_flag; //文件标志  int \_file; //文件的有效性验证  int \_charbuf; //检查缓冲区状况,如果无缓冲区则不读取  int \_bufsiz; //文件的大小  char \*\_tmpfname;//临时文件名  }; |

注意：文件在磁盘中的存储形式有两种：ASC文件和非ASC文件

1. 文件读写

|  |
| --- |
|  |

## 配置文件读写

1. 接口的封装与设计
2. 写配置文件
3. 读配置文件
4. 修改配置文件

注意:在软件开发中，接口要求紧，模块要求送

## 文件读写C语言API库函数

快速占领，完成自己人生财富库

|  |
| --- |
| 按字符读写文件 : fgetc() fputc()  按行读写文件: fgets() fputs() （配置读写）  按块读写文件: fread() ,fwirte()（大数据块迁移） |

Fopen() 打开方式：文件是否存在，重新清空写，文件尾部追加读写

|  |
| --- |
| r 以只读方式打开文件，该文件必须存在。  r+ 以可读写方式打开文件，该文件必须存在。  rb+ 读写打开一个二进制文件，允许读写数据，文件必须存在。  rw+ 读写打开一个文本文件，允许读和写。  w 打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。  w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。  a 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF符保留）  a+ 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。 （原来的EOF符不保留）  wb 只写打开或新建一个二进制文件；只允许写数据。  wb+ 读写打开或建立一个二进制文件，允许读和写。  ab+ 读写打开一个二进制文件，允许读或在文件末追加数据。  at+ 打开一个叫string的文件，a表示append,就是说写入处理的时候是接着原来文件已有内容写入，不是从头写入覆盖掉，t表示打开文件的类型是文本文件，+号表示对文件既可以读也可以写 |

Char\*fgets(char\*buf,int Maxlen,File\*fp);

## 大数据加解密

# 链表

## 动态链表的基本操作

|  |
| --- |
| SLIST \*SList\_Creat(); ----------------------链表创建返回值方式  int SList\_Creat2(SLIST \*\*mypHead)-----函数参数建链表的方式---内存第三种模式  int SList\_Print(SLIST \*pHead);------------打印输出链表节点的数据  int SList\_NodeInsert(SLIST \*pHead, int x, int y);--通过x的值，插入一个节点为y的值  int SList\_NodeDel(SLIST \*pHead, int x);----------删除节点为y的值  int SList\_Destory(SLIST \*pHead);------------销毁节点 |
| 链表建立第一种方式 SLIST \*SList\_Creat()-------  {  SLIST \*pHead = NULL, \*pM =NULL, \*pCur = NULL;  int data = 0;  //1 创建头结点并初始化  pHead = (SLIST \*)malloc(sizeof(SLIST));  if (pHead == NULL)  {  return NULL;  }  pHead->data = 0;  pHead->next = NULL;  //2 从键盘输入数据，创建业务结点  printf("\nplease enter the data of node(-1:quit) ");  scanf("%d", &data);  //3 循环创建  //初始化当前节点，指向头结点  pCur = pHead;  while(data != -1)  {  //新建业务结点 并初始化  //1 不断的malloc 新的业务节点 ===PM  pM = (SLIST \*)malloc(sizeof(SLIST));  if (pM == NULL)  {  SList\_Destory(pHead); //  return NULL;  }  pM->data = data;  pM->next = NULL;  //2、让pM节点入链表  pCur->next = pM;  //3 pM节点变成当前节点  pCur = pM; //pCur = pCur->next;  //2 从键盘输入数据，创建业务结点  printf("\nplease enter the data of node(-1:quit) ");  scanf("%d", &data);  }  return pHead;  } |
| 链表建立第二种方式： int SList\_Creat2(SLIST \*\*mypHead) // 2年工作经验那  {  int ret = 0;  SLIST \*pHead = NULL, \*pM =NULL, \*pCur = NULL;  int data = 0;  //1 创建头结点并初始化  pHead = (SLIST \*)malloc(sizeof(SLIST));  if (pHead == NULL)  {  ret = -1;  printf("func SList\_Creat2() err:%d ", ret);  return ret;  }  pHead->data = 0;  pHead->next = NULL;  //2 从键盘输入数据，创建业务结点  printf("\nplease enter the data of node(-1:quit) ");  scanf("%d", &data);  //3 循环创建  //初始化当前节点，指向头结点  pCur = pHead;  while(data != -1)  {  //新建业务结点 并初始化  //1 不断的malloc 新的业务节点 ===PM  pM = (SLIST \*)malloc(sizeof(SLIST));  if (pM == NULL)  {  SList\_Destory(pHead); //  ret = -2;  printf("func SList\_Creat2() err:%d ", ret);  return ret;  }  pM->data = data;  pM->next = NULL;  //2、让pM节点入链表  pCur->next = pM;  //3 pM节点变成当前节点  pCur = pM; //pCur = pCur->next;  //2 从键盘输入数据，创建业务结点  printf("\nplease enter the data of node(-1:quit) ");  scanf("%d", &data);  }  \*mypHead = pHead;  return ret;  } |

## 链表插入