# 01 内存四区专题讲座

## 01接口封装和设计思想引导

|  |
| --- |
| 对自己的要求：   1. 对自己的知识进行储备，理解所学的知识。 2. 能够准确的、管理自己的知识体系 3. 对每个事务的认识都是有规律的，要尊重规律。多花时间思考。 4. 关注文件的所有信息，创建时间，文件类型等。 |
| 1. 理解分层思想 2. 更好的封装接口 |

## 02排序热身及数组做函数参数

|  |
| --- |
| 1. 冒泡排序 2. 快速排序 3. 选择排序 |

## 03数据类型本质

|  |
| --- |
| 如**int a;**  从编辑器的角度来说，就是分配4个字节的内存，且这个内存的别名是a；因此数据类型的本质就是**固定内存**。注意是固定的。 |

## 04变量本质

|  |
| --- |
| 普通变量：如int a；内存模型就是分配**4个字节的连续内存**  如int \*p；指针变量存放地址，根据操作系统的位数大小分配内存，32位4字节，64位8字节。依然是连续的内存空间   1. 可通过变量对内存进行修改a = 10; 2. 通过指针间接对内存修改 \*p = 10; 3. 通过引用对内存修改（c++中），&a = 10；   变量总结：   1. 可对内存进行读写 2. 是往变量对应的内存中读写，而不是往变量里读写 3. 变量存放在代码区 |

## 05内存四区

|  |
| --- |
| 包括全局区、代码区、栈、堆。  四区 |

## 06指针铁律1：指针是一种数据类型

|  |
| --- |
| - 指针是一种**数据类型**，这个数据类型就是其**指向的内存空间分配时的数据类型**   1. 指针变量指向谁就把谁的地址赋值给指针变量 2. 没有内存，就没有指针，内存是一切的基础   - 指针也是一个**变量**，用来保存地址的 指针==地址  - 1.使用变量对内存直接进行读写操作  - int a; a= 10;  - 2.使用指针间接对内存进行读写操作  - int \*p; int a; p = &a; \*p = 10;  - \*(谁的地址)就是间接对其对应的内存进行读写操作  - \*p在=**左边是写内存**， \*p放在等号的**右边是读内存**  - 3.指针变量和他指向的内存空间不一样，例如，int a = 10; int \*p = &a;  - 有以下几个含义，例如  - p = 0x5511；只是改变了**指针变量**的值，完全不会影响指针所指**的内存里的数值**  - 给\*p赋值只会改变指针指向内存的值，不会改变指针变量的值  - int \*p; 执行的数据类型就是int 则**步长**就是sizeof(int);  - 4.**指针就是一个存放4个字节地址的变量，啥也不是 ，不过是几个字节，当初先\*的时候，只要一个就足够，他就是存放4个字节的指针，只有在使用的时候才关心它指向的内容究竟是什么。**  - 站在c++/c语言的编译器去看指针，指针就是一个变量，就是一个存放地址的变量，可能4或者8个字节  - 当c++编译器看到**一个** ‘ \* ’的时候就认为它是指针，就会分配4或者8个字节  - 当我们**使用的时候，采取考虑指针指向的内容是几维的**  - 永远记住：c语言规定**数组名**代表**数组首元素**的地址，如果对数组取地址&，就代表整个数组  - 5.使用\*p作为参数，在被调函数里面使用这个指针修改内存时，一定要保证指向的**内存能够修改**。比如指针指向常量区（全局区），修改其空间内的数据时，是不允许的，程序会出现错误。\*\***不要轻易改变指针输入特性中in内存块的内存**\*\* ，一般不修改 |

# 02 经验话语

|  |
| --- |
| 1.**一次偶然的成功，比必然的失败更可怕**  2.从编译器的角度去思考语法现象  3.概念不清晰和概念清晰但是训练不到位是产生bug的根源。一定要做到**概念清晰，训练到极致。** |

## 01 多级指针做函数参数的理解

|  |
| --- |
| 二级指针做函数参数，也是一个4个字节的地址，无其他。多级指针做函数参数的时候，不管是一个\* **还是8个**\*我们只需要知道编译器只给他跟配4个字节的内存即可。只有在使用的时候才去关心内容是什么。 |

## 02 C和java堆栈的区别

|  |
| --- |
| c和java的最大区别，在内存分配时，c语言可以在栈区（临时区）分配内存，而java则不行（应该也是可以的 栈区有内存存在的） |

## 03 [ ] 和 \* 的本质

|  |
| --- |
| Int a[10];  其中通过a[i]来访问数组a中的数值a[i]==>\*(a+i);  [] , \*都是取此时地址对应内存里面的数值。 |

## 04 为什么int a[10],a是个常量

|  |
| --- |
| 如果a不是常量发生改变，a数组的头部信息会发生变化，在系统对int a[10]；释放内存的时候是根据free list的头部信息的指针释放内存的。改变了就不能回收能存。a可以认为是一个常量指针，**为了避免内存释放失败c++编译器，不允许直接a++的操作。可以使用int \*p =a;p++的操作来遍历内存中的数值**  **- int a[10]; 使用下标访问就是利于理解**  **- a[10] --> a[0+i]-->a(0+i)-->\*(a+i)** |

## 05 栈和堆的生长方向

|  |
| --- |
| 4 |

# 03 字符串专题

## 01指针铁律2：间接赋值是指针存在的最大意义

|  |
| --- |
| 1. 指针间接赋值的条件 2. 定义形参或者实参变量 3. 建立形参和实参之间的关系，将实参的地址赋给形参 4. 通过形参间接的去修改实参对应的内存内容   三个条件，可以任意组合，着重理解**内存的分配**和**已分配内存的生命周期**   1. 应用场景 2. 在同一个函数中修改某个内存的值 3. 在不同的函数**之间**修改某个内存的值 4. 代码如下：   void getLen(int \*p)  {  \*p = 200; //3.形参间接修改实参变量的值  }  void test06(void)  {  int a; //1.定义一个变量--通常是实参  a = 10; //通过内存别名，变量改变内存数据  printf("直接改变a = %d \n", a);  int \*p; //1.定义一个变量通常是实参  p = &a; //2.建立关联  \*p = 100; //通过地址改变  printf("间接改变和访问\*p = %d \n", a);  getLen(&a); //通过实参取地址传入  printf("函数调用后a = %d \n", a);  }  **3.指针间接修改内存的值**  1级指针间接修改0级指针的值int a; int \*p = &a; \*p = 10;  2级指针间接修改1级指针的值 int a; int \*p = &a; int \*\*pp; pp = p; \*p = 10; //依次类推 |
| 二级指针修改一级指针  //p2就是一个变量 他来接收实参的地址  //如果实参是0级 就用1级 如果是1级就用二级  //\*就是一把钥匙  int getMem22(char \*\*p2)  {  \*p2 = 200; //  \*p2 = (char \*)malloc(100); //申请一个内存并将这个内存的地址分配给p2  return 0;  }  char \*\*getMem();//也行 但是不好，占用了函数返回状态  void main(void)  {  //一级指针变量的值 需要用二级指针变量来修改  char \*p = NULL;  //直接修改p的值 p是指针所以是存放的值的意义就是地址  p = 0x1;  p = 0x2;  printf("%d\n",p);  getMem22(&p);  printf("%d\n",p);  system("pause");  } |
| 三级指针间接修改二级指针  //通过三级指针修改二级指针  int modifyP(char \*\*\*p3) //  {  int i=0;  \*p3 = 200;  char \*\*temp = (char \*\*)malloc(10 \*sizeof(char\*));  memset(temp,0,10\*sizeof(char\*));  for(i=0; i<3; i++)  {  temp[i] = (char \*)malloc(100);  }  \*p3 = temp;  return 0;  }  void main(void)  {  char \*\*p2 = NULL;  char \*\*\*p3= NULL;  p2 = 1;  p2 = 3;  printf("p2: %d \n",p2);  p3 = &p2; //通过p3间接修改  \*p3 = 100;  printf("p2: %d\n",p2);  modifyP(&p2);  printf("p2: %d\n",p2);  } |

## 02字符串操作基础（使用一级指针）

|  |
| --- |
| **查找主串中子串存在的个数**  int getCountStr(char \*str,char \*target,int \*count)  {  char \*s = str;  char \*t = target;  int ncout = 0;  int ret = 0;  //要先判断错误 不要判断正确的之后再去写业务代码 很难维护  if(s == NULL || t == NULL || count == NULL)  {  ret = -1;  printf("func getCountStr check s,t,cout is err: %d\n",ret);  return ret;  }  do  {  s = strstr(s,t);  if(s == NULL) //找不到就直接返回  {  break;  }  else  {  ncout ++ ;  s = s + strlen(t);  }  }while(s != '\0');  \*count = ncout;  return ret;  } |
| **字符串复制程序的演变，需要注意的是复制的目标空间是不是以’\0’结尾。**  int copyStr1(char \*from ,char \*to)  {  char \*p1 = from;  char \*p2 = to;  int ret = 0;  if(p1 == NULL || p2 == NULL)  {  ret = -1;  printf("func copyStr1 is err: %d\n",ret);  return ret;  }  for(; \*p1!='\0'; p1++,p2++)  {  \*p2 = \*p1;  }  \*p2 = '\0';  return ret;  }  int copyStr2(char \*from ,char \*to)  {  char \*p1 = from;  char \*p2 = to;  int ret = 0;  if(p1 == NULL || p2 == NULL)  {  ret = -1;  printf("func copyStr1 is err: %d\n",ret);  return ret;  }  while(\*p1!='\0')  {  \*p2++ = \*p1++;  }  \*p2 = '\0';  return ret;  }  int copyStr3(char \*from ,char \*to)  {  char \*p1 = from;  char \*p2 = to;  int ret = 0;  if(p1 == NULL || p2 == NULL)  {  ret = -1;  printf("func copyStr1 is err: %d\n",ret);  return ret;  }  while((\*p2++ = \*p1++)!='\0');  return ret;  } |

## 03字符串内存模型

|  |
| --- |
| 字符串**模型底层**就是**字符数组**，均是以**’\0’**结尾，如果一个字符串的输出后面是乱码，可能是因为没有以’\0’结尾。Char buf[10] = {0}; //就表明里面都是’\0’；copy到这里是不需要添加’\0’,strcpy不会复制’\0’需要自**己手动添加**’\0’; 所以申请内存时，要多申请一个。   1. char \*p = “123”; //p是字符数组的首地址，指针变量存放在栈区，字符串存放到全局区，当已经存在这个字符串时，只需要将其首地址返回给指针即可. 2. char buf1[] = “abc”; //buf表示字符数组的首元素地址，字符串放再全局区，且有一份拷贝到了栈区，自动推到内存的大小。 3. char buf2[10] = “456”; //buf表示字符数组的首元素地址，字符串放再全局区，且有一份拷贝到了栈区，内存的大小为10。 4. malloc内存时如果存放的是字符串，而是通过strlen确定长度的，一定在malloc申请内存的时候增加1，不然容易冲刷掉其他内存。   char \*p = (char \*)malloc(strlen("abc")); //并没有**申请到'\0'**这个结束符的内存  char \*p =(char \*)malloc(strlen("abc")+1);  strcpy(p,"abc"); //会出现错误   1. 其中1和2最大的区别就是，2会在栈区复制一份字符串。**以上三种的内存中图如下** |

## 04指针铁律3：深入理解指针必须结合内存四区的概念，图和意境结合

|  |
| --- |
| 1. 主调函数分配内存，被调函数分配内存 2. 内存分配方式，1，指针作为输入，2,指针作为输出 3. 看函数声明时，一定要想的事，特别是指针作为参数时 4. - int copyStr2(char \*from ,char \*to，int \*len) 5. - 参数是由主调函数传入，则主调函数是不是应该**分配内存空间** 6. - 主调函数分配的内存空间的生命周期是否在被调函数未完成前被占用或者释放 7. - 被调函数接收到的参数，要进行判断是否正确 8. - 参数中那几个是**传入参数（in）**，那几个是需要**返回参数（输出out）**。如from，to均是传入参数。len则可能是需要返回的参数，也就是在给调函数里面进程逻辑处理之后的处理结果通过地址修改，传给主调函数。为什么不用返回值，因为，**返回值**只能有**一个参数**，而返回参数可能是多个，为啥不用数组呢？？ 数组的使用最后也是退化成指针。 |
| 1. 数组指针的规定：   int a[10]; sizeof(a) == 40个字节 4\*10 这是规定，不用去理解，而sizeof(&a) = 4;//首元素的地址  C语言规定&a代表整个数组，而&a+1 跳40个单元 这是规定 |

## 05被调用函数分配内存如何传出内存两种方法

|  |
| --- |
| 1. 通过return返回内存首地址   不能返回函数的调用状态了  char \*getM() //  {  char \*temp = NULL;  temp = (char \*)malloc(10);  char buf[] = "123";  strcpy(temp,buf);  return temp;  } |
| 1. 通过二级指针作为输出   可以返回函数运行的各个状态  int getM2(char \*\*p/\*out\*/)  {  int ret = 0;  if(p == NULL)  {  ret = -1;  printf("p is NULL\n");  return ret;  }  char \*temp;  temp = NULL;  temp = (char \*)malloc(10);  if(temp == NULL)  {  ret = -1;  printf("temp malloc is failed\n");  return ret;  }  strcpy(temp,"acb");  \*p = temp;  return ret;  }  //函数的调用  int main(void)  {  char \*p1 = NULL;  char \*p2 = NULL;  p1 = getM();  printf("%s\n",p1);  getM2(&p2); //二级指针间接修改一级指针的值  printf("%s\n",p2);  return 0;  } |
| 1. 调用函数时，参数时指针时，内存分配工作 2. **主调函数中，全局、栈、堆、分配的内存都可以作为输入给到被调函数。** 3. **被调函数只能将分配在堆、全局区的内存输出到主调函数，被主调函数使用** |
| 4. 调用函数是，参数时指针时，内存分配工作  申请堆一定要注意释放。在能够满足项目要求的情况下，尽量**不要使用堆**，能用buf就用buf，以免在释放时出现错误。 |
| 1. 应该根据函数的定义来揣测，是**主调函数**分配内存还是被调函数分配内存 2. 如int **getM**(char \*p1/\*in\*/,char \*\*p2/\*out\*/,int \*num/\*in\*/);可以看出p1可能作为输入,则主调函数分配内存，p2可以是被调函数分配内存，然后作为输出在主调函数中接收。num是主调函数修改。 |

## 06去除字符串两端的空格

|  |
| --- |
| //主调函数分配内存，并多开辟一个内存存放处理好的字符串  //若是在输入的内存上直接保存，就要注意这个指针指向的内存是否能被修改  //使用buf和指针模型  //两个功能差不多的函数是不是可以使用函数指针  //需要参数相同才可以  int trimSpaceStr(char \*from ,char \*to)  {  int ret = 0;  int i,j,len;  if(from == NULL || to == NULL)  {  ret = -1;  printf("from or to is NULL,err:%d\n",ret);  return ret;  }  char \*f = from;  char \*t = to;  len = strlen(f);  i = 0;  j = len-1;  while(isspace(f[i]) && i<j) i++;  while(isspace(f[j]) && j>0) j--;  len = j-i+1; //剩余的长度  //strncpy(t,f[i],len); //不行，f[i]属于元素内容  strncpy(t,f+i,len); //f+i 表示地址  t[len] = '\0';  return ret;  }  //使用原来的内存作为更改后的存储地址  //1.这个空间能更改  int trimSpaceStr\_1(char \*from)  {  int ret = 0;  int i,j,len;  if(from == NULL)  {  ret = -1;  printf("from or to is NULL,err:%d\n",ret);  return ret;  }  char \*f = from;  len = strlen(f);  i = 0;  j = len-1;  while(isspace(f[i]) && i<j) i++;  while(isspace(f[j]) && j>0) j--;  len = j-i+1; //剩余的长度  //strncpy(t,f[i],len); //不行，f[i]属于元素内容  strncpy(f,f+i,len); //f+i 表示地址  f[len] = '\0';  return ret;  }  int main(void)  {  int ret = 0;  //char src[] = " d afq "; //这里可以用src接收 因为他复制到栈里去了  char \*src = " d afq "; //不能使用本身 因为指向的内存是全局，全局区不能修改  char det[1024];  //ret = trimSpaceStr(src,det);  ret = trimSpaceStr\_1(src);  if(ret != 0)  {  printf("trimSpaceStr return value is error,err:%d\n",ret);  return ret;  }  printf("%s\n",det);  printf("%s\n",src);  return ret;  } |

## 07两个辅助指针变量挖字符串

|  |
| --- |
|  |

## 08字符串反转

|  |
| --- |
| //可以传入，也可以不传入字符串长度  //不传入字符串长度就需要传入的字符串是以'\0'结尾的  //int revStr(char \*p,int len)  int revStr(char \*p)  {  int ret = 0;  int slen = 0;  char \*left = NULL;  char \*right = NULL;  if(p == NULL)  {  ret = -1;  printf("p args is empty,err:%d\n",ret);  return ret;  }  left = p;  slen = strlen(left); //传长度 和不传长度  right = p + slen - 1;  char tmp;  while(left<right) //结束条件是left == right  {  tmp = \*left;  \*left = \*right;  \*right = tmp;  left++;  right--;  }  return ret;  } |

## 09项目开发中易错模型的建立

|  |
| --- |
| 1.申请内存时，尽量保证有足够的内存存放数据，可以预留一定的内存  2.计算好边界问题，避免早晨内存被冲刷的问题  void test03(void)  {  char \*r = NULL;  char \*s1 = "123";  char \*s2 = "456";  //如果不+1 内存不会直接进行报错的 但是不知何时就会出现问题  //**内存紧张 可能堆内存会被冲刷掉**  **r = (char \*)malloc(strlen(s1) + strlen(s2)+1);**  if(r == NULL)  {  printf("malloc is failed\n");  exit(1);  }  strcpy(r,s1); //易错语句  strcat(r,s2); //易错语句时  printf("%s\n",r);  free(r);  r = NULL;  } |
| 1. 字符串修改，字符指针和字符数组的区别。 2. 字符指针在全局区，字符数组在全局区和栈区，修改的是栈区，内存模型参考**字符串内存模型**   void test04(void)  {  char \*p = NULL,\*q = NULL;  char buf1[10];  char \*buf2;  // buf1 = "xyz"; //不能赋值 只能通过strcpy  strcpy(buf1,"123");  // buf2 = buf1; //也不能这样操作char buf2[10];  buf2 = buf1;  buf2[1] = '4';  \*(buf2+2) = '5';  printf("%s,%s\n",buf1,buf2); //145 145 复制到栈里面去了。。。且buf2指向buf1  //因为在全局区所以不能改  //=======================  p = "xyz";  q = p;  //q[1] = 'Y'; //执行报错  //\*(q+1) = 'Y'; //是不能这样的,报错  printf("%s\n",q);  printf("%s\n",p);  } |
| 关于首地址增加的不同,对于堆中可以修改首内存的指针，不会报错，但是在进行free的时候将不能free申请的内存，因为c编译器free是根据首地址里面的指针进行free的。  而对于数组，也是根据首地址free的，并且数组元素不放在堆中，是放到全局和栈中的，首地址是一个常量，不能被修改。想操作地址变化只能用一个同类型的指针变量进行接收。  char \*p = NULL;  char buf[20] ;  char \*q = NULL;  p = (char \*)malloc(100);  q = p;  p++; //可以 如果要操作最好使用一个变量接收然后操作，保留首地址内存  p = p + 2; //可以  **free(p); //此时不能free申请到的内存**  **Free(q); //k可以完成释放**  //buf++; //不可以，  buf = buf + 1; //不可以 |
| **实参传给形参时，多个形参保存的内存地址是连续的**  int falut1(int \*nums,int numsSize)  {  int i=0;  int \*p = nums;  for(i=0; i<numsSize; i++)  {  \*p++ = i+2;  }  printf("%d\n",numsSize);  \*p = 0; //此时就修改了主调函数里面的值了，因为主调内存分配是连续的  //导致主调函数的len = 0了;  }  int main()  {  int i=0;  int p[] = {1,2,3,4,5};  int len = sizeof(p)/sizeof(p[0]);  falut1(p,len); //\*p 最后越界操作了 导致len对应的内存被修改  for(i=0; i<len ;i++ )  {  printf("%d ",p[i]);  }  return 0;  } |

# 04 二级指针输入模型专题

## 01多维数组名的本质

|  |
| --- |
| **数组名是一个地址是一个不可变的常量地址，可使用一个变量指针接收，进行操作**  int a[10];  a表示这个数组第一个元素的地址，和&a[0]的地址一样的，但是，步长的是根据**数组类型决定。**  int a[3][10];  a表示这个数组的第一行第一列元素的首地址，和&啊a[0][0]的地址一样，但是在计算步长时则存在区别。  a[0]表示这个数组第一行第一个元素的地址（相当于一级指针）  a和a[0]返回的地址相同，但是a+1,a[0]+1的步长是不一样的，a步长为10\*4，也就是指向的一维数组的长度，而a[0]的步长时此时一维数组里元素的字节数4  printf("%d\n",a); //行地址 a+1,a+2..  printf("%d\n",a[0]); //元素地址 a[0]+1....  printf("%d\n",a[0][0]); //元素值  访问a[1][2]的退化成指针的过程为  a[1][2]->(\*(a+1))[2]->\*(\*(a+1)+2) |
| **测试程序**  int main(void)  {  int a[3][3] = {{0,1,2},{3,4,5},{6,7,8}};  printf("%d\n",a); //6356708  printf("%d\n",a[0]); //6356708  printf("%d\n",a+1); //6356720 多12  printf("%d\n",a[0]+1); //6356712 多4  printf("%d\n",a[1][2]); //5  printf("%d\n",\*(\*(a+1)+2)); //5  return 0;  } |

## 02多维数组做函数参数退化问题

|  |
| --- |
| **1.c语言中，当基本数据类型做函数参数时**，主要是值传递，也就是将实参的值拷贝一份给形参，但是对于数组和字符串等复杂数据类型，c语言编译器会将函数参数退化成指针的形式  如:  Int fun(int a[10],int b) ;  会将a[10]==>转成地址a（数组的首元素地址）  **原因1**：高效  **原因2:**不用拷贝主调函数里面分配的内存了，只需要根据首地址+偏移地址即可访问a所指向的空间内容 |
| **2.二维数组退化过程**   1. 二维数组可以看成是一维数组，不过其中的每个元素都是一维数组 2. 二维数组做参数时，第一维数据可以省略 3. 虽然数组作为参数时会退化，直接第一指针可能会效率高一点，其实效率并没有什么差别，经过编译器优化之后。**建议使用指针接收**   Void fun(int a[10]) ==>void fun(int a[])==> void fun(int\* a)  Void fun(int a[3][10])==>void fun(int a[][10])==>void fun(int (\*a)[10]) |
| **3.一维和二维数组和指针的对应关系**  一维数组 char a[30] 一级指针 char \*a  指针数组 char \*a[30] 指针的指针 char \*\*a  二维数组 char a[10][30] 数组指针 char (\*a)[30] |

## 03二级指针三种内存模型建立

|  |
| --- |
| 三种二级指针的模型表示方式   1. char \*pszBuf[10]; 2. char pszBuf[3][10]; 3. char \*\*pszBuf; |
| **在内存中的对应关系如下（见到就要想到这张图）**  模型2 |

## 04二级指针输入模型概念

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <string.h>  int getArray3Free(char \*\*p3, int num)  {  int i=0;  char \*\*tmp = p3;  if(tmp == NULL)  {  //free(p3); //垃圾语句  return 0;  }  for(i=0; i<num; i++)  {  free(tmp[i]);  tmp[i] = NULL;  }  free(tmp);  tmp = NULL;  return 0;  }  char \*\*getArray3(char \*\*mp1,int num1,char (\*mp2)[30],int num2,int \*num3)  {  int i=0,j=0;  int tempNum3 =0;  char buf[30];  char \*\*tmp3 = NULL;  if(mp1 == NULL || mp2 == NULL)  {  printf("in is failed\n");  return NULL;  }  tempNum3 = num1 + num2;  //分配第一维  tmp3 = (char \*\*)malloc((tempNum3+1)\*(sizeof(char \*)));  if(tmp3 == NULL)  {  printf("malloc is failed!\n");  return NULL;  }  //分配第二维  for(i=0; i<num1; i++)  {  tmp3[i] = (char \*)malloc(strlen(mp1[i])+1);  }  for(j=0; j<num2; j++,i++)  {  tmp3[i] = (char \*)malloc(strlen(mp2[j])+1);  }  tmp3[i] = '\0';  //拷贝  for(i=0; i<num1; i++)  {  strcpy(tmp3[i],mp1[i]);  }  for(j=0; j<num2; j++,i++)  {  strcpy(tmp3[i],mp2[j]);  }  //排序  for(i=0; i<tempNum3; i++)  {  for(j=0; j<tempNum3-i-1; j++)  {  if(strcmp(tmp3[j],tmp3[j+1])>0)  {  strcpy(buf,tmp3[j]);  strcpy(tmp3[j],tmp3[j+1]);  strcpy(tmp3[j+1],buf);  }  }  }  \*num3 = tempNum3;  return tmp3;  }  /\*  将模型1 和 模型2中的内容复制到模型三里面去 并且输出内容  \*/  int mainReturn(void)  {  int num3 = 0, i=0;  char \*p1[] = {"2222","3333","1111"}; //模型1  //printf("%d\n",strlen("aaaa"));  char p2[4][30] = {"bbbb","aaaa","zzzz","ccc"}; //模型2  char \*\*p3 = NULL;  p3 = getArray3(p1,3 ,p2,4 ,&num3);  if(p3 == NULL)  {  return -1;  }  for(i=0; i<num3; i++)  {  printf("%s ",p3[i]);  }  getArray3Free(p3,num3);  return 0;  }  //不使用返回值作为输出了 则可以使用多级指针作为输出  //间接修改二级指针就需要使用三级指针  int getArray4(char \*\*mp1,int num1,char (\*mp2)[30],int num2,char \*\*\*mp3,int \*num3)  {  int i=0,j=0;  int ret = 0;  char \*tt = NULL;  char \*\*temp = NULL;  int tempLen = num1+num2;  //这里就是判断地址mp3对应的内存里面的值了 值是就可以为0的而地址不能直接这样做 \*0肯定会出问题  //if(\*mp3 == NULL)  if(mp1 == NULL || mp2 == NULL || mp3 == NULL)  {  ret = -1;  printf("in memo is err: %d\n",ret);  }  temp = (char \*\*)malloc((tempLen+1)\*sizeof(char \*));  for(i=0; i<num1; i++)  {  temp[i] = (char \*)malloc(strlen(mp1[i])+1);  }  for(j=0; j<num2; j++,i++)  {  temp[i] = (char \*)malloc(strlen(mp2[j])+1);  }  temp[i] = '\0';  for(i=0; i<num1; i++)  {  strcpy(temp[i],mp1[i]);  }  for(j=0; j<num2; j++,i++)  {  strcpy(temp[i],mp2[j]);  }  for(i=0; i<tempLen; i++)  {  for(j=0; j<tempLen-i-1; j++)  {  if(strcmp(temp[j],temp[j+1])>0)  {  tt = temp[j];  temp[j] = temp[j+1];  temp[j+1] = tt;  }  }  }  \*mp3 = temp;  free(temp);  \*num3 = tempLen;  return 0;  }  int getArray4Free(char \*\*\*p3,int num3)  {  int i=0;  char \*\*temp = NULL;  if(p3 == NULL)  {  return -1;  }  temp = \*p3;  for(i=0; i<num3; i++)  {  if(temp[i] != NULL)  {  free(temp[i]);  }  }  \*p3 = NULL; //通过间接赋值直接减p3设置为NULL 所以在主调函数里面就不需要再设置成NULL了  return 0;  }  int main31(void)  {  int num3 = 0, i=0;  char \*p1[] = {"2222","3333","1111"};  //printf("%d\n",strlen("aaaa"));  char p2[4][30] = {"bbbb","aaaa","zzzz","ccc"};  char \*\*p3 = NULL;  //易犯错误1  // char \*\*\*p4 = NULL;  // //p4直接传入 ，NULL = 0 所以在函数中对\*p4赋值的时候是肯定会出现错误的，不能直接将这样传参数  // getArray4(p1,3 ,p2,4 ,p4,&num3);  getArray4(p1,3 ,p2,4 ,&p3,&num3);  if(p3 == NULL)  {  return -1;  }  for(i=0; i<num3; i++)  {  printf("%s ",p3[i]);  }  // getArray3Free(p3,num3);  // p3 = NULL;  getArray4Free(&p3,num3);  return 0;  } |

## 05第一种和第三种二级指针模型的对比

|  |
| --- |
| Char \*p[]; //第一种  Char p[3][10]; //第三种吗 |
| #include <stdlib.h>  #include <stdio.h>  #include <string.h>  //交换变量的数值时有两种方法  //1.更改变量的指向 2.直接更改变量指向的内存里面的内容  //分配堆内存  char \*\*getMemory(int num)  {  int i=0;  //内存中存放的还是地址  char \*\*temp = (char \*\*)malloc(num \* sizeof(char \*));  for(i=0; i<num; i++)  {  temp[i] = (char \*)malloc(100); //每个分配100个字节  }  return temp; //由于是在堆中申请的 因此可以返回给主调函数使用  }  //在申请内存的是时候多申请一个，在处理这个内存的时候可以不用传长度的  char \*\*getMemory01(int num)  {  int i=0;  char \*\*tep = (char \*\*)malloc((num+1) \* sizeof(char \*));  for(i=0; i<num; i++)  {  tep[i] = (char \*)malloc(100);  }  tep[num] = '\0'; //结束符  // tep[num] = NULL;  // tep[num] = 0;  return tep;  }  void sortArray03(char \*\*myArray)  {  int i=0,j=0;  char \*p;  for(i=0;myArray[i]!='\0'; i++ )  {  for(j=i+1;myArray[j]!='\0' ; j++)  {  if(strcmp(myArray[i],myArray[j])>0)  {  p = myArray[i];  myArray[i] = myArray[j];  myArray[j] = p;  }  }  }  }  int testNoLen(void)  {  int i=0;  char \*\*p = NULL;  p = getMemory01(3); //这里多申请了一个存放结束符的内存，如果不申请，程序运行的时候直接挂掉  strcpy(p[0],"bbb");  strcpy(p[1],"ccc");  strcpy(p[2],"aaa");  for(i=0; i<3; i++)  {  printf("%s\n",p[i]);  }  printf("=====================\n");  sortArray03(p);  for(i=0; i<3; i++)  {  printf("%s\n",p[i]);  }  return 0;  }  void sortArray(char \*\*myArray,int num)  {  int i=0,j=0;  char \*temp;  for(i=0; i<num; i++)  {  for(j=i+1; j<num; j++)  {  //直接更改指针的指向  if(strcmp(myArray[i],myArray[j])>0)  {  temp = myArray[i];  myArray[i] = myArray[j];  myArray[j] = temp;  }  }  }  }  void sortArray02(char \*\*myArray,int num)  {  char temp[200];  int i=0,j=0;  for(i=0; i<num; i++)  {  for(j=i+1; j<num; j++)  {  if(strcmp(myArray[i],myArray[j])>0)  {  //直接操作内存  strcpy(temp,myArray[i]);  strcpy(myArray[i],myArray[j]);  strcpy(myArray[j],temp);  }  }  }  }  int test02(void)  {  char \*\*p = NULL;  int i=0;  p = getMemory(3); //申请内存  strcpy(p[0],"bbbb");  strcpy(p[1],"aaaa");  strcpy(p[2],"cccc");  printf("before: \n");  for(i=0; i<3; i++)  {  printf("%s \n",p[i]);  }  sortArray02 (p,3);  printf("after: \n");  for(i=0; i<3; i++)  {  printf("%s \n",p[i]);  }  //释放内存  for(i=0; i<0; i++)  {  free(p[i]);  }  free(p);  return 0;  }  int main15(void)  {  printf("二级指针1和3模型的比较\n");  test02();  //testNoLen();  return 0;  } |

## 06数组类型、数组指针类型、数组指针类型变量（重要，使用typedef实现）

|  |
| --- |
| 数组类型  void arrType(void)  {  //通过typedef 来定义数组指针  typedef int myArrayType[5]; //int  int i;  myArrayType myArr; //int myArr[5];  for(i=0; i<5; i++)  {  myArr[i] = i+1;  }  for(i=0; i<5; i++)  {  printf("%d\n",myArr[i]);  }  printf("=============================\n");  //数组指针  myArrayType \*p=NULL; //定义一个数组类型的指针 指向数组  int a[5];  p = &a; //p就相当于二级指针 a表示地址 &表示a的地址的那个地址  for(i=0; i<5; i++)  {  //加\*之后，变成一级指针  (\*p)[i] = i+1; // a[i] = i+1;  }  for(i=0; i<5; i++)  {  printf("%d\n",(\*p)[i]);  }  //可以直接定义一个数组指针不过这个指针的  int (\*b)[5];  //int \*b = NULL; 虽然也存放地址 但是意义不一样 不能这样通过指针间接访问  b = &a; //能不能改变呢  for(i=0; i<5; i++)  {  printf("%d\n",(\*b)[i]);  }  } |
| 数组指针类型定义讲解  void pType(void)  {  typedef int arrType[5];  int i=0;  //定义一个数组指针类型  //定义一个指针类型， 定义一个指向数组的指针  typedef int (\*pArrType)[5]; // 二级指针了  pArrType myP; //int (\*myp)[5];  int b[5];  myP = &b; //变量去地址给指针  for(i=0; i<5; i++)  {  (\*myP)[i] = i+1;  }  for(i=0; i<5; i++)  {  printf("%d\n",(\*myP)[i]);  }  } |
| 使用typedef可能太麻烦，也可以直接定义  //使用typedef太麻烦 可以直接定义  **数组指针类型变量，用来保存数组的首地址的**  void myPointer(void)  {  int a[5]; //数组类型  int (\*p)[5]; //数组指针类型 p数组指针类型变量==  int i=0;  p = &a;  for(i=0; i<5; i++)  {  (\*p)[i] = i+1;  }  for(i=0; i<5; i++)  {  printf("%d\n",(\*p)[i]);  }  printf("=============\n");  for(i=0; i<5; i++)  {  printf("%d\n",a[i]);  }  } |

## 08多维数组做函数参数退化原因剖析

|  |
| --- |
|  |

## 09三种二级指针模型作为输入考试应用

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <string.h>  int getArray3Free(char \*\*p3, int num)  {  int i=0;  char \*\*tmp = p3;  if(tmp == NULL)  {  //free(p3); //垃圾语句  return 0;  }  for(i=0; i<num; i++)  {  free(tmp[i]);  tmp[i] = NULL;  }  free(tmp);  tmp = NULL;  return 0;  }  char \*\*getArray3(char \*\*mp1,int num1,char (\*mp2)[30],int num2,int \*num3)  {  int i=0,j=0;  int tempNum3 =0;  char buf[30];  char \*\*tmp3 = NULL;  if(mp1 == NULL || mp2 == NULL)  {  printf("in is failed\n");  return NULL;  }  tempNum3 = num1 + num2;  //分配第一维  tmp3 = (char \*\*)malloc((tempNum3+1)\*(sizeof(char \*)));  if(tmp3 == NULL)  {  printf("malloc is failed!\n");  return NULL;  }  //分配第二维  for(i=0; i<num1; i++)  {  tmp3[i] = (char \*)malloc(strlen(mp1[i])+1);  }  for(j=0; j<num2; j++,i++)  {  tmp3[i] = (char \*)malloc(strlen(mp2[j])+1);  }  tmp3[i] = '\0';  //拷贝  for(i=0; i<num1; i++)  {  strcpy(tmp3[i],mp1[i]);  }  for(j=0; j<num2; j++,i++)  {  strcpy(tmp3[i],mp2[j]);  }  //排序  for(i=0; i<tempNum3; i++)  {  for(j=0; j<tempNum3-i-1; j++)  {  if(strcmp(tmp3[j],tmp3[j+1])>0)  {  strcpy(buf,tmp3[j]);  strcpy(tmp3[j],tmp3[j+1]);  strcpy(tmp3[j+1],buf);  }  }  }  \*num3 = tempNum3;  return tmp3;  }  /\*  将模型1 和 模型2中的内容复制到模型三里面去 并且输出内容  \*/  int mainReturn(void)  {  int num3 = 0, i=0;  char \*p1[] = {"2222","3333","1111"};  //printf("%d\n",strlen("aaaa"));  char p2[4][30] = {"bbbb","aaaa","zzzz","ccc"};  char \*\*p3 = NULL;  p3 = getArray3(p1,3 ,p2,4 ,&num3);  if(p3 == NULL)  {  return -1;  }  for(i=0; i<num3; i++)  {  printf("%s ",p3[i]);  }  getArray3Free(p3,num3);  return 0;  }  //不使用返回值作为输出了 则可以使用多级指针作为输出  //间接修改二级指针就需要使用三级指针  int getArray4(char \*\*mp1,int num1,char (\*mp2)[30],int num2,char \*\*\*mp3,int \*num3)  {  int i=0,j=0;  int ret = 0;  char \*tt = NULL;  char \*\*temp = NULL;  int tempLen = num1+num2;  //这里就是判断地址mp3对应的内存里面的值了 值是就可以为0的而地址不能直接这样做 \*0肯定会出问题  //if(\*mp3 == NULL)  if(mp1 == NULL || mp2 == NULL || mp3 == NULL)  {  ret = -1;  printf("in memo is err: %d\n",ret);  }  temp = (char \*\*)malloc((tempLen+1)\*sizeof(char \*));  for(i=0; i<num1; i++)  {  temp[i] = (char \*)malloc(strlen(mp1[i])+1);  }  for(j=0; j<num2; j++,i++)  {  temp[i] = (char \*)malloc(strlen(mp2[j])+1);  }  temp[i] = '\0';  for(i=0; i<num1; i++)  {  strcpy(temp[i],mp1[i]);  }  for(j=0; j<num2; j++,i++)  {  strcpy(temp[i],mp2[j]);  }  for(i=0; i<tempLen; i++)  {  for(j=0; j<tempLen-i-1; j++)  {  if(strcmp(temp[j],temp[j+1])>0)  {  tt = temp[j];  temp[j] = temp[j+1];  temp[j+1] = tt;  }  }  }  \*mp3 = temp;  free(temp);  \*num3 = tempLen;  return 0;  }  int getArray4Free(char \*\*\*p3,int num3)  {  int i=0;  char \*\*temp = NULL;  if(p3 == NULL)  {  return -1;  }  temp = \*p3;  for(i=0; i<num3; i++)  {  if(temp[i] != NULL)  {  free(temp[i]);  }  }  \*p3 = NULL; //通过间接赋值直接减p3设置为NULL 所以在主调函数里面就不需要再设置成NULL了  return 0;  }  int main31(void)  {  int num3 = 0, i=0;  char \*p1[] = {"2222","3333","1111"};  //printf("%d\n",strlen("aaaa"));  char p2[4][30] = {"bbbb","aaaa","zzzz","ccc"}; //不能直接使用\*\*p来接收  char \*\*p3 = NULL;  //易犯错误1  // char \*\*\*p4 = NULL;  // //p4直接传入 ，NULL = 0 所以在函数中对\*p4赋值的时候是肯定会出现错误的，不能直接将这样传参数  // getArray4(p1,3 ,p2,4 ,p4,&num3);  getArray4(p1,3 ,p2,4 ,&p3,&num3);  if(p3 == NULL)  {  return -1;  }  for(i=0; i<num3; i++)  {  printf("%s ",p3[i]);  }  // getArray3Free(p3,num3);  // p3 = NULL;  getArray4Free(&p3,num3);  return 0;  } |

## 10第三种内存模型结束标志

|  |
| --- |
| 18 |
| 二级指针的模型图解 |
| 二级指针模型 |

## 11野指针产生原因和解决方案

|  |
| --- |
| #include <string.h>  #include <stdio.h>  #include <stdlib.h>  **//1.定义指针时，现将其指向NULL**  **//2.申请内存之后，先判断是否申请成功在使用**  **//3.释放内存先判断，释放之后让其为NULL;**  //可以成功申请内存  char \*getM1(int num)  {  char \*tmp = NULL;  tmp = (char \*)malloc(100);  return tmp;  }  //不利用返回 利用指针作为参数的应用场景  void getM2(int count , char \*p) //思考内存场景就能知道  {  char \*tmp = NULL;  tmp = (char \*)malloc(100);  p = tmp; //其实并没有改变实参 也就接收这个函数的指针依然是NULL  }  //使用指针间接赋值  void getM3(int count , char \*\*p)  {  char \*tmp = NULL;  tmp = (char \*)malloc(100);  \*p = tmp; //直接通过地址间接修改实参的值  }  //释放内存  void freeM(char \*p)  {  if(p != NULL)  {  free(p); // 只是释放了实参和形参指向的同一个内存块的内存  p = NULL; //这里并不能让实参的值为NULL  }  }  int main23(void)  {  /\*  char \*p = NULL;  p = (char \*)malloc(100);  free(p);  p = NULL;  \*/  char \*p = NULL;  getM2(10,p); //p作为实参依然是NULL  if(p == NULL)  {  printf("p == NULL\n");  }  getM3(10,&p);  strcpy(p,"123");  printf("%s\n",p);  if(p==NULL)  {  printf("p == NULL\n");  }  else  {  printf("p != NULL\n");  }  freeM(p); //确实free了但是为什么 还能二次free  p = NULL;  printf("free after");  printf("%s\n",p); //输出了乱码  if(p==NULL)  {  printf("p == NULL\n");  }  else  {  printf("p != NULL\n");  }  freeM(p); //并没有报错 按道理来说是不是应该报错  return 0;  } |

## 12 const

|  |
| --- |
| //两者意义完全相同  char const a;  const char a;  char \* const p; //表示地址变量的内容不能被改变，也就是指向不能被改变，但指向的内容可以被改变  char const \* p; //表示指针指向的内容不能被改变，指针可以改变  const char const \* p; //指向和指向的内容均不能被改变  //表示指针指向的内容不能被改变，指针可以改变  int get1(const char \*p)  {  p[0] = 1;//错误正确  p=0x01;  return 0;  }  //表示地址变量的内容不能被改变，也就是指向不能被改变，但指向的内容可以被改变  int get2(char \* const p)  {  p[0] = 5; //正确  p = 0x02; //错误  return 0;  }  //指向和指向的内容均不能被改变  int get2(char \* const p)  {  p[0] = 5; //错误  p = 0x02; //错误  return 0;  }  int main(void)  {  return 0;  } |

## 13函数中一个入口，多个返回（出口），释放内存

|  |
| --- |
| 1.//在每一个出口前，将申请的内存空间进行释放  int main(void)  {  int ret = 0;  ....  char \*p1 = getMem(10);  ...  if(ret != 0) //多个返回 多个出口  {  free(p1);  return -1;  }  char \*p2 = getMem(20);  if(ret != 0)  {  free(p1);  free(p2);  return -1; //出口  }  free(p1);  free(p2);  return 0;  } |
| 2.//当内存释放过于繁琐的时候可以使用goto ，使用goto一定是要出口，不能作为进入  int main(void)  {  int ret = 0;  ....  char \*p1 = getMem(10);  ...  if(ret != 0)  {  goto End;  }  char \*p2 = getMem(20);  if(ret != 0)  {  goto End;  }  End:  if(p1!=NULL)  free(p1);  if(p2 != NULL)  free(p2);  return 0;  } |

## 14 第三种内存模型强化

|  |
| --- |
|  |

# 05 结构体专题

## 01点操作和指针操作的本质

|  |
| --- |
| **->**和**.**操作只是获得相对于结构体的偏移地址 a->name a.age在cpu中运行，并没有操作内存，只有用=时才操作指针  结构体分配内存都是捆绑分配，也就是一个结构体分配的内存会分别分配给自己的**成员变量**和**成员函数** |

## 02结构体做函数参数

|  |
| --- |
| **结构体做函数参数初级**  typedef struct \_Teacher  {  char name[62];  char \*title;  char c;  int age;  } Teacher;  int printTArray(Teacher \*tArray,int num)  {  int i=0;  for(i=0; i<num; i++)  {  printf("%d %s %s\n",tArray[i].age,tArray[i].name,tArray[i].title);  }  return 0;  }  int sortTArray(Teacher \*tArray,int num)  {  int i=0,j=0;  Teacher tep; //**临时区分配内存进行内存块的操作**  for(i=0; i<num-1; i++)  {  for(j=0; j<num-i-1; j++)  {  if(tArray[j].age > tArray[j+1].age)  {  tep = tArray[j]; //直接使用编译器提供的**结构体复制**行为  tArray[j] = tArray[j+1];  tArray[j+1] = tep;  }  }  }  return 0;  }  //使用分配到**堆里面的内存** satck操作 结构体  //这种场景是不能修改实参的  int createTArray(Teacher \*tArray,int num) //这里是形参 如果  {  tArray = (Teacher \*)malloc(num\* sizeof(Teacher)); //分配内存  if(tArray == NULL)  {  return -1;  }  return 0; //  }  //这种场景是可以修改实参的,缺点是不能返回，这个函数的状态  Teacher \*createTArray01(int num)  {  int i=0;  Teacher \*temp = NULL;  temp = (Teacher \*)malloc(num \* sizeof(Teacher));  if(temp == NULL)  {  return NULL;  }  for(i=0; i<num; i++)  {  temp[i].title = (char \*)malloc(100); //分配num个内存块  }  return temp;  }  int main(void)  {  int i=0;  //int ret = 0;  Teacher \*pArray = NULL; //分配到heap区,  **//使用一级指针不能直接修改一级指针的内容**  //ret = createTArray(pArray,3); //pArray 和 tArray 一个是形参和实参是不一样，因此出现**pArray是空的**  pArray = createTArray01(3); //pArray 和 tArray 一个是形参和实参是不一样，因此出现pArray是空的  if(pArray == NULL)  {  printf("pArray = NULL\n");  return 0;  }  for(i=0; i<3; i++)  {  printf("please enter %d age value:", i+1);  scanf("%d",&pArray[i].age);  printf("please enter %d name value:", i+1);  scanf("%s",pArray[i].name);  printf("please enter %d title value:", i+1);  scanf("%s",pArray[i].title);  }  printf("sort before..........\n");  printTArray(pArray,3);  sortTArray(pArray,3);  printf("sort before..........\n");  printTArray(pArray, 3);  free(pArray);  pArray = NULL;  return 0;  } |

## 03结构体中套一级指针

|  |
| --- |
| 结构体做函数参数进阶（结构体套一级指针），要注意里面指针指向的内存申请和释放  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  //结构体里面套指针是学习的重点  typedef struct \_Teacher  {  char name[62];  char \*title;  char c;  int age;  } Teacher;  int printTArray(Teacher \*tArray,int num)  {  int i=0;  for(i=0; i<num; i++)  {  printf("%d %s %s\n",tArray[i].age,tArray[i].name,tArray[i].title);  }  return 0;  }  int sortTArray(Teacher \*tArray,int num)  {  int i=0,j=0;  Teacher tep; //临时区分配内存进行内存块的操作  for(i=0; i<num-1; i++)  {  for(j=0; j<num-i-1; j++)  {  if(tArray[j].age > tArray[j+1].age)  {  tep = tArray[j]; //直接使用编译器提供的结构体复制行为  tArray[j] = tArray[j+1];  tArray[j+1] = tep;  }  }  }  return 0;  }  //使用分配到堆里面的内存 satck操作 结构体  //这种场景是不能修改实参的  int createTArray(Teacher \*tArray,int num) //这里是形参 如果  {  tArray = (Teacher \*)malloc(num\* sizeof(Teacher)); //分配内存  if(tArray == NULL)  {  return -1;  }  return 0; //  }  //这种场景是可以修改实参的,缺点是不能返回，这个函数的状态  Teacher \*createTArray01(int num)  {  int i=0;  Teacher \*temp = NULL;  temp = (Teacher \*)malloc(num \* sizeof(Teacher));  if(temp == NULL)  {  return NULL;  }  for(i=0; i<num; i++)  {  **//给结构体嵌套的一级指针申请内存**  temp[i].title = (char \*)malloc(100); //分配num个内存块  }  return temp;  }  int **freeP**(Teacher \*parr,int numSize)  {  int i=0;  if(parr == NULL)  {  return -1;  }  Teacher \*temp = parr;  for(i=0; i<numSize; i++)  {  if(temp[i].title != NULL)  {  free(temp[i].title);  temp[i].title = NULL;  }  }  return 0;  }  int main(void)  {  int i=0;  //int ret = 0;  Teacher \*pArray = NULL; //分配到heap区,  //ret = createTArray(pArray,3); //pArray 和 tArray **一个是形参和实参是不一样，**因此出现pArray是空的  pArray = createTArray01(3); //pArray 和 tArray 一个是形参和实参是不一样，因此出现pArray是空的  if(pArray == NULL)  {  printf("pArray = NULL\n");  return 0;  }  for(i=0; i<3; i++)  {  printf("please enter %d age value:", i+1);  scanf("%d",&pArray[i].age);  printf("please enter %d name value:", i+1);  scanf("%s",pArray[i].name);  printf("please enter %d title value:", i+1);  scanf("%s",pArray[i].title);  }  printf("sort before..........\n");  printTArray(pArray,3);  sortTArray(pArray,3);  printf("sort before..........\n");  printTArray(pArray, 3);  **//这里貌似没有释放结构体里的一级指针对应的内存 可以自己定义一个函数释放内存**  free(pArray);  pArray = NULL;  return 0;  } |

## 04结构体中套二级指针

|  |
| --- |
| **//结构体里面套指针是学习的重点**  typedef struct \_Teacher  {  char name[64];  **char \*title;**  **char \*\*pStuArray;**  int age;  } Teacher;  int printTArray(Teacher \*tArray,int num)  {  int i=0,j=0;  for(i=0; i<num; i++)  {  printf("老师信息: ");  printf("%d %s %s\n",tArray[i].age,tArray[i].name,tArray[i].title);  printf("学生信息: ");  for(j=0; j<3; j++)  {  **printf("%s ", tArray[i].pStuArray[j]);**  }  printf("\n");  }  return 0;  }  int sortTArray(Teacher \*tArray,int num)  {  int i=0,j=0;  Teacher tep; //临时区分配内存进行内存块的操作  for(i=0; i<num-1; i++)  {  for(j=0; j<num-i-1; j++)  {  if(tArray[j].age > tArray[j+1].age)  {  tep = tArray[j]; //直接使用编译器提供的结构体复制行为  tArray[j] = tArray[j+1];  tArray[j+1] = tep;  }  }  }  return 0;  }  //使用分配到堆里面的内存 satck操作 结构体  //这种场景是不能修改实参  //这种场景是可以修改实参的,缺点是不能返回，这个函数的状态  Teacher \*createTArray01(int num)  {  int i=0;  Teacher \*temp = NULL;  temp = (Teacher \*)malloc(num \* sizeof(Teacher));  if(temp == NULL)  {  return NULL;  }  for(i=0; i<num; i++)  {  temp[i].title = (char \*)malloc(100); //分配num个内存块  }  for(i=0; i<num; i++)  {  //先申**请内存 再将内存的首地址挂到需要分配的内存指针上**  //这样结构清晰明了  **char \*\*ptmp = (char \*\*)malloc(3\*sizeof(char \*));**  for(int j=0; j<3; j++)  {  **ptmp[j] = (char \*)malloc(100);**  }  temp[i].pStuArray = ptmp;  }  return temp;  }  int freeArray(Teacher \*tArray,int num)  {  int i=0,j=0;  if(tArray == NULL)  {  return -1;  }  for(i=0; i<num; i++)  {  char \*\*tep = tArray[i].pStuArray;  if(tep==NULL)continue;  for(j=0; j<3; j++)  {  if(tep[j]!=NULL)  {  free(tep[j]);  }  }  free(tep);  }  for(i=0; i<num; i++)  {  free(tArray[i].title);  tArray[i].title = NULL; //垃圾语句 因为形参不能改变实参  }  free(tArray);  tArray = NULL;  return 0;  }  int main(void)  {  int i=0,j=0;  //int ret = 0;  Teacher \*pArray = NULL; //分配到heap区,然后  //ret = createTArray(pArray,3); //pArray 和 tArray 一个是形参和实参是不一样，因此出现pArray是空的  pArray = createTArray01(3); //pArray 和 tArray 一个是形参和实参是不一样，因此出现pArray是空的  if(pArray == NULL)  {  printf("pArray = NULL\n");  return 0;  }  for(i=0; i<3; i++)  {  printf("请输入第%d个老师的年龄:", i+1);  scanf("%d",&pArray[i].age);  printf("请输入第%d个老师的姓名:", i+1);  scanf("%s",pArray[i].name);  printf("请输入第%d个老师的职称", i+1);  scanf("%s",pArray[i].title);  printf("学生信息: \n");  for(j=0; j<3; j++)  {  printf("请输入第%d学生的姓名: ",j+1);  scanf("%s",pArray[i].pStuArray[j]);  }  }  printf("sort before..........\n");  printTArray(pArray,3);  sortTArray(pArray,3);  printf("sort after..........\n");  printTArray(pArray, 3);  //free(pArray);  freeArray(pArray,3);  pArray = NULL;  return 0;  } |

## 05编译器浅拷贝操作

|  |
| --- |
|  |

## 06深拷贝和浅拷贝

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  typedef struct \_ATeacher  {  char \*name;  int age;  } Teacher;  Teacher \*createT()  {  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);  }  }  //浅拷贝，编译器只会复制对应位置的指针到变量中，内存是不会帮我们复制的  //当结构体中有buf时 是没有问题的 但是当里面有指针的时候c++编译器只会进行指针变量复制  //编译器不会多分配内存  //当在释放一次内存之后，再去通过指针去释放这个内存的时候会出现错误  //使用函数进行手动深拷贝  int copyObj(Teacher \*to,Teacher \*from)  {  //\*to = \* from;  memcpy(to,from,sizeof(Teacher));  to->name = (char \*)malloc(100);  strcpy(to->name,from->name);  return 0;  }  int main19(void)  {  Teacher t1;  Teacher t2;  t1.name = (char \*)malloc(100);  t1.age = 10;  //t2 = t1; //按道理会报错但是没有  copyObj(&t2,&t1);//调用自己写的函数完成深拷贝  if(t1.name != NULL)  {  free(t1.name);  }  if(t2.name != NULL)  {  free(t2.name);  }  return 0;  } |

# 06 C操作文件专题

## 01文件读写

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int main22(void)  {  FILE \*fp = NULL;  char \*fname = "1.txt";  char \*t = "123";  char buf[100];  fp = fopen(fname,"w+");  if(fp == NULL)  {  printf("fopen is failed.\n");  }  fputs(t,fp);  fgets(buf,20,fp);  printf("%s\n",buf);  printf("hhh\n");  if(fp != NULL)  {  fclose(fp);  }  return 0;  } |
| 按行读文件1 |
| FREAD |

## 02文件读写需要注意的事项

|  |
| --- |
|  |

# 07函数指针

## 01固定参数函数指针

|  |
| --- |
| int add(int a,int b)  {  return a+b;  }  int sub(int a,int b)  {  return a - b;  }  int mul(int a) //需要具有相同的参数  {  return 2\*a;  }  int main(void)  {  int (\*p)(int a,int b); //根据参数来定义函数指针  p = add; //函数指针变量接收的是函数名，函数的地址就是函数的地址  int ret = 0;  ret = (\*p)(1,2);  printf("ret = %d\n",ret);  p = sub;  ret = (\*p)(1,2);  printf("ret = %d\n",ret);  } |

## 02可变参数的函数指针

|  |
| --- |
| int add(int a)  {  return a+2;  }  int sub(int a,int b)  {  return a - b;  }  int main(void)  {  int ret = 0;  int (\*p)(); //c中无参就可以接收多个参数，所以可以接收多个不同的函数  p = add;  ret = (\*p)(2);  printf("%d\n",ret);  p = sub;  ret = (\*p)(3,1);  printf("sub = %d\n",ret);  return 0;  } |

# 08链表

## 01单链表

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  typedef struct Node  {  int data;  struct Node \*next;  }SLIST;  //删除链表 释放链表的所有节点  int SLIST\_Destory(SLIST \*pHead)  {  SLIST \*p = NULL,\*temp = NULL;  if(pHead == NULL)  {  printf("destory is failed\n");  return -1;  }  p = pHead;  while(p)  {  temp = p->next;  free(p);  p = temp;  }  return 0;  }  //2年工作经验  SLIST \*SLIST\_create()  {  SLIST \*pHead = NULL,\*pCur=NULL,\*pMal = NULL;  int data;  pHead = (SLIST \*)malloc(sizeof(SLIST));  if(pHead == NULL)  {  printf("malloc failed \n");  return NULL;  }  pHead->data = 0;  pHead->next = NULL;  //pCur = (SLIST \*)malloc(sizeof(SLIST));  printf("please input data:\n");  scanf("%d",&data);  //pHead->next = pCur;  pCur = pHead;  while(data != -1)  {  pMal = (SLIST \*)malloc(sizeof(SLIST));  if(pMal == NULL) //当申请某个节点失败之后，前面的节点要全部释放 不然造成内存泄漏  {  //可以增加日志  SLIST\_Destory(pHead);  return NULL;  }  pMal->data = data;  pMal->next = NULL;  pCur->next = pMal;  pCur = pMal;  printf("please input data:\n");  scanf("%d",&data);  }  return pHead;  }  int SLIST\_create2(SLIST \*\*myHead)  {  SLIST \*pHead = NULL,\*pCur=NULL,\*pMal = NULL;  int data;  pHead = (SLIST \*)malloc(sizeof(SLIST));  if(pHead == NULL)  {  printf("malloc failed \n");  return NULL;  }  pHead->data = 0;  pHead->next = NULL;  printf("please input data:\n");  scanf("%d",&data);  pCur = pHead;  while(data != -1)  {  pMal = (SLIST \*)malloc(sizeof(SLIST));  if(pMal == NULL) //当申请某个节点失败之后，前面的节点要全部释放 不然造成内存泄漏  {  //可以增加日志  SLIST\_Destory(pHead);  return NULL;  }  pMal->data = data;  pMal->next = NULL;  pCur->next = pMal;  pCur = pMal;  printf("please input data:\n");  scanf("%d",&data);  }  \*myHead = pHead;  return 0;  }  int SLIST\_print(SLIST \*pHead)  {  SLIST \*p = NULL;  if(pHead == NULL)  {  printf("pHead failed\n");  return -1;  }  p = pHead->next;  printf("begin\n");  while(p!=NULL)  {  printf("%d ",p->data);  p = p->next;  }  printf("\nend\n");  return 0;  }  //在值为x的节点前插入一个节点  int SLIST\_Insert(SLIST \*pHead,int x,int y)  {  SLIST \*pPre=NULL,\*pCur=NULL;  SLIST \*pMal = NULL;  if(pHead == NULL)  {  return -1;  }  pPre = pHead;  pCur = pHead->next;  pMal = (SLIST \*)malloc(sizeof(SLIST));  pMal->data = y;  pMal->next = NULL;  while(pCur)  {  if(pCur->data == x)  {  break;//出去进行交换  }  pPre = pCur;  pCur = pCur->next;  }  pMal->next = pCur;  pPre->next = pMal;  return 0;  }  //删除数值为x的节点  int SLIST\_Del(SLIST \*pHead,int x)  {  SLIST \*pPre = NULL,\*pCur = NULL;  if(pHead == NULL)  {  return -1;  }  pPre = pHead;  pCur = pHead->next;  while(pCur)  {  if(pCur->data == x)  {  pPre->next = pCur->next; //找到了就删除  free(pCur);  break;  }  pPre = pCur;  pCur = pCur->next;  }  //没找到不用操作  return 0;  }  int SLIST\_Reverse(SLIST \*pHead)  {  SLIST \*p=NULL,\*q=NULL,\*t=NULL;  if(pHead == NULL)  {  return -1;  }  if(pHead->next == NULL || pHead->next->next == NULL)  {  return -2;  }  p = pHead->next;  q = pHead->next->next;  while(q)  {  t = q->next;  q->next = p;  p = q;  q = t;  }  pHead->next->next = NULL;  pHead->next = p;  return 0;  }  int main33(void)  {  SLIST \*p = NULL;  //p = SLIST\_create();  SLIST\_create2(&p); //使用二级指针创建内存 被调用函数分配内存  SLIST\_print(p);  SLIST\_Insert(p,2,20);  SLIST\_print(p);  SLIST\_Del(p,3);  SLIST\_print(p);  SLIST\_Reverse(p);  printf("反转之后\n");  SLIST\_print(p);  SLIST\_Destory(p);  system("pause");  return 0;  } |

## 02传统链表pk非传统链表

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  //传统节点不可能包含所有业务模型  //那么就利用非传统节点，每个业务模型都包含一个相应的节点，根据这个节点串联所有  //的业务模型数据  typedef struct Node2  {  int data;  char name[60];  char num[60];  struct Node2 \*next;  }SLIST;  //链表和业务数据的数据域没有完成独立分离  //链表完成不了所有的业务数据  //想组织一个链表的业务节点，把所有的业务模型都包含..（问题的产生原因）  typedef struct node  {  struct node \*next;  }node;  //让这个业务模型去包含业务节点  //Teacher不管怎么变都和链表没有任何关系  //linux内核链表就是这种形式  typedef struct \_Teacher3  {  int age;  char name[60];  char \*\*p2;  struct node myNode;  }Teacher;  int main(void)  {  system("pause");  return 0;  }  linux非传统链表 |

## 03企业级链表库调用的例子

|  |
| --- |
| #include<stdio.h>  #include <string.h>  #include <stdlib.h>  //自己定义的不算数的  typedef struct Node1  {  struct Node1 \*next;  }LinkListNode;  //业务结构体  //即使修改业务结构体 只需要在业务结构体中加入NOde即可 所以说  //业务代码和链表库没有什么关系  typedef struct \_Teacher4  {  //必须在业务节点的第一个域包含链表节点  LinkListNode node;  char name[32];  int age;  }Teacher;  /\*  //链表库提供的api接口 如何去调用  int LinkList;  LinkList \*LinkList\_Create();  void LinkList\_Destory(LinkList\* list);  void LinkList\_Clear(LinkList\* list);  int LinkList\_Length(LinkList\* list);  int LinkList\_Insert(LinkList\* list,LinkListNode \*node,int pos);  LinkListNode \*LinkList\_Get(LinkList \*list,int pos);  LinkListNode \*LinkList\_Delete(LinkList \*list,int pos);  \*/  int main34(void)  {  int ret = 0;  int len = 0;  int i=0;  LinkList \*list = NULL;  Teacher t1,t2,t3,t4,t5;  memset(&t1,0,sizeof(Teacher));  memset(&t2,0,sizeof(Teacher));  memset(&t3,0,sizeof(Teacher));  memset(&t4,0,sizeof(Teacher));  memset(&t5,0,sizeof(Teacher));  t1.age = 1;  t2.age = 2;  t3.age = 3;  t4.age = 4;  t5.age = 5;  list = LinkList\_Create();  if(list == NULL)  {  return -1;  }  //插入节点 使用尾插法  //内存申请了得 所以可以直接转  //这里也是转换的是指针类型 指针类型由于内存空间大小都是一样的所以可以直接抢转  ret = LinkList\_Insert(list,(LinkListNode \*)&t1,LinkList\_Length(list));  ret = LinkList\_Insert(list,(LinkListNode \*)&t2,LinkList\_Length(list));  ret = LinkList\_Insert(list,(LinkListNode \*)&t3,LinkList\_Length(list));  ret = LinkList\_Insert(list,(LinkListNode \*)&t4,LinkList\_Length(list));  ret = LinkList\_Insert(list,(LinkListNode \*)&t5,LinkList\_Length(list));  len = LinkList\_Length(list);  for(i=0; i<len; i++)  {  //由于Get的返回值是ListNode和也就是这个节点的首地址  //而Teacher的内存的第一个域就是ListNode所以可以直接转成Teacher  struct \_Teacher4 \*tmp = (struct \_Teacher4\*)LinkList\_Get(list,i);  if(tmp != NULL)  {  printf("age : %d\n",tmp->age);  }  }  //删除  while(LinkList\_Length(list))  {  //从链表库中删除业务节点的时候，把业务节点的指针返回给调用者，  //以便调用者进行额外的逻辑控制  struct \_Teacher4 \*tmp = (struct \_Teacher4 \*)LinkList\_Delete(list,0); //头删  if(tmp != NULL)  {  printf("age : %d\n",tmp->age);  }  }  LinkList\_Destory(list);  return 0;  } |

# 09 C语言框架

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | 专题 | 知识点组成 | 面试 | 工作 | 备注 |
| 01 | 内存四区 | 排序热身及数组做函数参数 | \*\* | \*\*\* |  |
| 数据类型本质、变量本质 | \*\* | \*\*\* |  |
| 内存四区模型和函数调用模型 | \*\* | \*\*\* | 全局区站位的点、临时区stack；栈向下生长、栈的生长方向和内存空间buf存放方向是两个不同的概念，堆向上生长的 |
| 接口封装设计思想引导 | 因人而异 | \*\*\* | 大方向 |
| 指针是一种数据类型 | \*\*\* | \*\*\* | 指针是变量，\*p操作内存，指针变量和它指向的内存块，指针补偿（p++）,指针是是一种数据类型，指向内存空间的数据类型 |
| 多级指针做函数参数的理解 | \* | \*\*\* | 角度1，2//多维数组做函数参数，一般情况，只能表达到2维，再多就没有意义了 |
| 02 | 铁律2：间接赋值试指针存在的最大意义 | 技术推演、间接赋值的应用场景  间接赋值成立条件，间接赋值推论，间接赋值的现实意义 | \* | \*\*\* |  |
| 明确研究方向，指针做函数参数 | \* | \*\*\* |  |
| 不断的改变指针的指向（不断给指针赋值，就是修改指针的指向） | \* | \*\*\* |  |
| 03 | 字符串专题 | 字符串基础 | \* | \*\*\* | C风格字符串是以’\0’结尾的字符串；初始化方法； |
| 字符串内存模型 | \*\* | \*\*\* | c语言字符串可以放在堆、栈、全局区 |
| strstr的while，do while模型 | \*\* | \*\*\* |  |
| 两头堵模型（两种写法，改指针改内存） | \*\* | \*\*\* |  |
| 字符串反转模型 | \*\* | \*\*\* |  |
| 两个辅助指针变量挖字符串 | \*\* | \*\*\* |  |
| 项目开发中一级指针易犯错误模型 | \* | \*\*\* |  |
| 初级const | \* | \*\*\* |  |
| 04 | 二级指针输入模型 | 第一种内存模型 | \*\* | \*\*\* |  |
| 第二种内存模型 | \*\* | \*\*\* |  |
| 第三种内存模型 | \*\* | \*\*\* |  |
| 多指针做函数参数退化问题 | \*\* | \*\*\* |  |
| 数组类型，数组指针类型，数组指针类型变量 | \*\* | \*\*\* | 压死初学者的三座大山 |
| 多维数组做函数参数退化原因剖析 | \*\* | \*\*\* |  |
| 二级指针三种内存模型建立 | \*\* | \*\*\* | 重点 |
| 第三种内存模型强化 | \*\* | \*\*\* | 交换指针变量和交换指针变量所指的内存空间的内容 |
| 第三种内存模型结束标志 | \*\* | \*\*\* |  |
| 野指针产生原因和解决方案 | \*\* | \*\*\* | 一级指针做函数参数，实参和形参是两个不同的概念 |
| 05 | 结构体专题 | 结构体类型定义及变量定义 | \* | \*\*\* |  |
| 深copy和浅copy | \* | \*\*\* |  |
| 结构体中套buf | \* | \*\*\* |  |
| 结构体中套一级指针和二级指针 | \* | \*\*\* |  |
| 06 | 文件操作专题 | 文件api基础 | \* | \*\*\* | 文件读写api，控制api |
| 配置文件读写库的设计实现 | \* | \*\*\* |  |
| 大数据文件加密解密实现 | \* | \*\*\* |  |
| 07 | 动态库的封装和设计 | 指针做输出的特性 | \*\* | \*\*\* | 玩转多级指针，易犯错误模型 |
| handle的意思，框架搭建及调试核心代码编写，日志功能集成 | \*\* | \*\*\* |  |
| 08 | 链表专题 | 结构体中套结构体 | \* | \*\*\* |  |
|  |  | 传统链表 | \*\* | \*\*\* | 链表创建、删除  插入、遍历、销毁、逆置 |
|  |  | 非传统链表/企业级链表 | \* | \*\*\* |  |
| 09 | 一级指针应用能力 | 不断强化 | \*\* | \*\*\* |  |

大端存储：高地址存高字节，低地址存低字节，高存高 低存低

小段存储：高地址存低字节，低地址存高字节

# C++语言的特点和联系

# 01 类需要注意的问题

## 01 构造函数

|  |
| --- |
| 1. 当类中不提供任何构造函数时，c++编译器自动提供无参构造和拷贝构造 2. 当类中提供任意非拷贝构造函数（有、无参）时，c++不会提供无参构造函数 3. 当类中定义拷贝构造函数时，c++不会提供构造函数 4. 默认提供的拷贝构造函数，只是对成员变量简单的赋值 |

## 02 c++编译器对普通函数的内部处理

|  |
| --- |
| cpp模型2 |

## 03 C++存储变量和函数

|  |
| --- |
|  |
| 模型3总结 |