# 01内存四区专题讲座

## 接口封装和设计思想引导

|  |
| --- |
| 接口封装设计思想引导  Sckclient客户端api模型设计 |
| 第一套api函数 |
| #ifndef \_SCK\_CLINT\_H\_  #define \_SCK\_CLINT\_H\_  //函数声明  // 1、客户端环境初始化  int sckClient\_init(void \*\*handle); //5 day  //  // 2、客户端发送报文  int sckClient\_send(void \*handle, unsigned char \*data, int datalen);  // 3、客户端端接受报文  int sckClient\_rev(void \*handle, unsigned char \*out, int \*outlen); //1  // 4、客户端环境释放  int sckClient\_destroy(void \*handle);  #endif |
| //条件编译 避免头文件多次包含  #ifndef \_SCK\_CLINT02\_H\_  #define \_SCK\_CLINT02\_H\_  #ifdef \_\_cplusplus  extern "C" {  #endif  //函数声明  // 1、客户端环境初始化  int sckClient\_init2(void \*\*handle); //5 day  //  // 2、客户端发送报文  int sckClient\_send2(void \*handle, unsigned char \*data, int datalen);  // 3、客户端端接受报文  int sckClient\_rev2(void \*handle, unsigned char \*\*out, int \*outlen); //1  int sckClient\_rev2\_Free(void \*\*p); //1  // 4、客户端环境释放  int sckClient\_destroy2(void \*\*handle);  #ifdef \_\_cplusplus  }  #endif  #endif |
| 我们找到了一套标准，我们可以高效、有目的的学习。 |
| Socket动态库业务模型思路分析 |
|  |

经验话语

|  |
| --- |
| Shift+del 删除一行 ctrl+shift+u大小 ctrl +u 小写  Alt+F9  F5在多个断点间切换 |

## 排序热身及数组做函数参数

|  |
| --- |
|  |
| //当数组当做函数参数的话的时候会退化为指针  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;  } |
| //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  //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 |
| 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;  } |
|  |

## 数据类型本质

|  |
| --- |
| 数据类型可理解为创建变量的模具（模子）；是固定大小内存的别名。 |
| sizeof是操作符，不是函数；sizeof测量的实体大小为编译期间就已确定  数据类型可以取别名、测量大小 |
| 数据类型的封装  Void数据类型的封装 |
| 数据类型的引申  C一维数组、二维数组有数据类型吗 3  C语言中，函数是可以看做一种数据类型吗？15 |
| 数组类型三大技术难点，压死初学者的三座大山 |

## 变量本质

|  |
| --- |
| 变量本质：（一段连续）内存空间的别名、内存空间的标号 |
| 修改变量的3种方法  1、直接  2、间接。内存有地址编号，拿到地址编号也可以修改内存；于是。。。横空出世了！  3、c++ 引用 |
| 总结：1对内存 可读可写； 2通过变量往内存读写数据，3不是向变量读写数据。4向变量代表的数据空间读写数据。变量跑到什么地方去了？ |

## 内存四区

|  |
| --- |
| 1. 内存四区模型和函数调用模型 |
| 基本概念 |
| 函数1调用函数2，函数1称为主调函数 函数2称为被调用函数 |
| 规则1：Main（主调函数）分配的内存（在堆区，栈区、全局区）都可以在被调用函数里使用吧。 |
| 规则2：在被调用函数里面分配的内存    1、如果在被调用函数里面的临时区（栈）分配内存，主调用函数是不能使用的。 |
| 全局区：//c++编译器优化  char \*getStr1()  {  char \*p = "abcd1";  return p;  }  char \*getStr2()  {  char \*p = "abcd1";  return p;  }  // |
| 临时区stack  char \* getStr3()  {  char buf[100];  memset(buf, 0, sizeof(buf));  strcpy(buf, "abcd1");  return buf;  } |
| //栈属性  //栈向下生长的，  //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，  //演示：stack生长方向  int main31()  {  float \*p1 = NULL;  int \*p2 = NULL;  int a = 0;  int b= 0;  char buf[16];  printf("&p1:%x, &p2:%x, &a:%x, &b:%x \n", &p1, &p2, &a, &b);  printf("&buf[0]:%x, &buf[1]:%x", &buf[0], &buf[1]);  getchar();  }  //软件开发中 注意野指针  //仔细观察malloc内存地址大小  //演示heap生长方向  int main32()  {  int a = 0;  int b = 0;  char \*p1 = NULL;  char \*p2= NULL;  p1 = (char \*)malloc(16);  p2 = (char \*)malloc(16);  printf("\n p1:%x, p2:%x", p1, p2);  printf("\n &p1:%x, &p2:%x", &p1, &p2);  //通过内存地址间接赋值  \*((char \*)0x394da0) = 'a';  \*((char \*)0x394da1) = 'b';  //通过内存地址间接修改内存空间的值  //通过变量名访问内存空间  //通过内存地址间接访问内存空间 这就是C语言的灵活性，也是c语言的精华  printf("\np2[0]:%c", p2[0]);  printf("\np2[1]:%c", p2[1]);  if (p1 != NULL)  {  free(p1);  }  if (p2 != NULL)  {  free(p2);  }  getchar();  return 0;  } |

## 铁律1：指针是一种数据类型

1. 指针也是一种变量，占有内存空间，用来保存内存地址

测试指针变量占有内存空间大小

2）\*p操作内存

在指针声明时，\*号表示所声明的变量为指针

在指针使用时，\*号表示 操作 指针所指向的内存空间中的值

\*p相当于通过地址(p变量的值)找到一块内存；然后操作内存

\*p放在等号的左边赋值（给内存赋值）

\*p放在等号的右边取值（从内存获取值）

3）指针变量和它指向的内存块是两个不同的概念

//含义1 给p赋值p=0x1111; 只会改变指针变量值，不会改变所指的内容；p = p +1; //p++

//含义2 给\*p赋值\*p='a'; 不会改变指针变量的值，只会改变所指的内存块的值

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

//含义4 =左边char \*p

//含义5 保证所指的内存块能修改

4）指针是一种数据类型，是指它指向的内存空间的数据类型

含义1：指针步长（p++），根据所致内存空间的数据类型来确定

p++=🡺(unsigned char )p+sizeof(a);

结论：指针的步长，根据所指内存空间类型来定。

# 02经验话语

## 01多级指针做函数参数的理解

|  |
| --- |
|  |
| //在函数调用哪个的时候 实参的值机械的传给形参（c int数组场景）  //关于形参：  写在函数上形参变量，还是写在函数里面的变量，  从CC++编译的角度来讲，是没有任何区别的（分配4字节内存）；  只不过是 写在函数上形参变量 ，具有对外的属性而已 |
| //数据类型分为两种，一个是简单的数据类型，一个是复杂的数据类型。碰见复杂的数据类型不能用简单的数据类型的思维去思考它。抛砖 |
| /\*  int getbuf01(char \*p); int getbuf01(char\* p);  int getbuf02(char \*\*p); int getbuf02(char \* \*p); getbuf02(char \*\* p);  int getbuf03(char (\*p)[]); int getbuf03(char (\*p) []); int getbuf03(char ( \*p)[ ]);  int getbuf03(char p[10][30]);  int getbuf04(char \*\*\*\*\*p);  \*/  //角度1站在c++编译器的角度 指针就是一个变量，除此之外啥也不是！  //不管是1个\* 还是8个\*对c++编译器来讲，只会分配4个字节内存  //角度2：当我们程序员要使用指针所指向的内存空间的时候，我们关心，这个内存块是一维的，还是二维的。  //一般情况：1级指针代表1维，二级指针代表二维。。。  //如果有超过char \*\*\*级及3级以上的指针，则不代表几维的内存。。。 |
| //多维数组做函数参数，一般情况下，只能表达到二维，  //如果是三维内存（我们程序员起的名字），已经没有意义。 |
| //证明一下多维数组的线性存储  //线性打印 |
| void printfAARRR(char \*\*\*ddd);  void printfAARRR(char \*\*\*\*\*\*\*\*\*dddd);  void printfArray411(int \*array, int num)  {  int i = 0;  for (i=0; i<num ; i++)  {  printf("%d ", array[i]);  }  }  void printfArray412(int (\*array)[5], int num)  {  return ;  }  void printfArrr333(int c[3][4][5])  {  return ;  }  void main()  {  int a[3][5];  int c[3][4][5];  int i , j = 0;  int tmp = 0;  for (i=0; i<3; i++)  {  for (j=0; j<5; j++)  {  a[i][j] = tmp ++;  }  }  printfArray411((int \*)a, 15);  system("pause");  } |
|  |

## 02C和java的堆栈区别

|  |
| --- |
| C可以在临时区分配内存块。。。。。。。java不行 |
| {  char \*p1 = 0; //  strcpy(p1, "abcdefg");  strcpy(0, "abcdefg"); //抛砖：在两个函数里面就不一定能明白  } |

## 03【】\*的本质

|  |
| --- |
| //[] \*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]; |
| //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");  } |

## 04为什么int a[10] a是个常量

|  |
| --- |
| {  int a[10]; //a是一个指针===》a常量指针===》为什么c++  int \*p = a;  p ++;  a ++;  }  //c++编译器要拿着a去析构内存，为了避免你把a的指向改变。。。。。 |

# 03铁律2\*p是指针存在的最大意义

|  |
| --- |
|  |
| 间接赋值成立的是3个条件 |
| /\* 间接赋值成立的三个条件  条件1 //定义1个变量（实参） //定义1个变量（形参）  条件2//建立关联：把实参取地址传给形参  条件3：//\*形参去间接地的修改了实参的值。  \*/  Int iNum = 0; //实参  int \*p = NULL;  p = &iNum;  iNum = 1;  \*p =2 ; //通过\*形参 == 间接地改变实参的值  \*p成立的三个条件： |
| 间接赋值成立三个条件的几种组合  123在一个函数里面  12 3 两个函数   1. 23两个函数   //间接赋值条件应用深入分析 三个条件的组合，分别产生三种很重要的语法现象  //123都写在一个函数里面  //12写在一个函数里面 3 写在另外一个函数里面  //1 写在一个函数里面 23 写在另外一个函数里面 抛砖。。。到时候别不认识啊。。。。。 |
| 间接赋值应用场景12  场景1：一个函数之内 \*p1++ = \*p2++  场景2：int getFileLen(int \*a ) |
| 间接赋值的推论  //在函数调用的时候  /\*  用1级指针形参，去间接修改了0级指针（实参）的值。。  用2级指针形参，去间接修改了1级指针（实参）的值。。  用3级指针形参，去间接修改了2级指针（实参）的值。。  用n级指针形参，去间接修改了n-1级指针（实参）的值。。  \*/ |
| 间接赋值的工程意义  //函数调用时，形参传给实参，用实参取地址，传给形参，在被调用函数里面用\*p，来改变实参，把运算结果传出来。  //指针作为函数参数的精髓。  //C语言特有的想象，是C语言的精华。。。 |
| 寻路  指针做函数参数是我们的研究重点。。。。。  指针是子弹、函数像枪管，，子弹枪管才能发挥它的威力。。。。。。。 |
| 下一步你的方向   1. 指针学完了。。。。。你只是c语言的半壁江山。。。。。 2. 函数指针。。。。 |
|  |
|  |
|  |

# 03字符串专题讲座

## 字符串操作基础

|  |
| --- |
| //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 |
| void main11()  {  //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[] = "abcdefg";  printf("%s\n", buf4 );  system("pause");  } |
| printf("%s\n", buf4 );  printf("sizeof(buf4): %d\n ", sizeof(buf4)); //注意sizeof是对数组类型进行大小测量 包括了\0  printf("strlen(buf4): %d \n", strlen(buf4));//strlen是求字符串的长度不包括\0 |
|  |
|  |

## 字符串内存模型

|  |
| --- |
| 一级指针内存模型图 |

字符串做函数参数

|  |
| --- |
| C库字符串API函数调用经验谈  字符串copy函数技术推演 |
|  |
| //C字符串函数调用方法经验谈  //站在内存四区模型和函数调用模型去思考函数。。。。。api接口  /\*  1） 主调函数 被调函数  a) 主调函数可把堆区、栈区、全局数据内存地址传给被调用函数  b) 被调用函数只能返回堆区、全局数据  2） 内存分配方式  a) 指针做函数参数，是有输入和输出特性的。  \*/ |
|  |

## 铁律3 深入理解指针必须和内存四区概念相结合，注意指针的输入输出特性

|  |
| --- |
| //C字符串函数调用方法经验谈  //站在内存四区模型和函数调用模型去思考函数。。。。。api接口  /\*  1） 主调函数 被调函数  a) 主调函数可把堆区、栈区、全局数据内存地址传给被调用函数  b) 被调用函数只能返回堆区、全局数据  2） 内存分配方式  a) 指针做函数参数，是有输入和输出特性的。  \*/  字符串操作常见工程开发模型  业务模型&业务测试模型分离===》接口封装和设计第一步 |

## 被调用函数分配内存如何传出 两种方法

|  |
| --- |
|  |
| //被调用函数分配内存吧结果甩出来有两种方法  //return  //指针做函数参数  char \* getBuffer()  {  char buf[109];  char \*p = (char \*)malloc(199);  //char \*p2= (char \*)malloc(199);  return p;  } |

## 项目开发中字符串模型建立

|  |
| --- |
| strstr的while dowhile模型 |
| //int cltClient\_rev(void \*handle, unsigned char \*buf, int \*buflen)  //不要相信别人给你传送的内存地址是可用的  int getCout(char \*str, char \*substr, int \*count)  {  int rv = 0;  char \*p = str;    int ncout = 0;  if (str==NULL || substr== NULL || count==NULL)  {  rv = -1;  printf("func getCout()check (str==NULL || substr== NULL || count==NULL) err:%d \n" , rv);  return rv;  }  do  {  p = strstr(p, substr);  if (p == NULL) //没有找到则跳出来  {  break;  }  else  {  ncout++;  p = p + strlen(substr);  }  } while (\*p != '\0');  //fuzhi  \*count = ncout;    printf("ncout:%d\n", ncout);  return rv;  } |
| void main36()  {  char \*p = "abcd1111abcd222abcd3333";  int ncout = 0;  while (p = strstr(p, "abcd"))  {  p = p + strlen("abcd");  ncout ++;  if (\*p == '\0')  {  break;  }  }  printf("ncout:%d\n", ncout);  system("pause");  } |
| 两头堵模型（两种写法） |
| //求去掉空格  //int trimSpaceStr2(char \*p, unsigned char \*buf2, int \*buf2len)  int trimSpaceStr2( char \*p, char \*buf2)  {  int ret = 0;  int ncount = 0;  int i, j;  i = 0;  j = strlen(p) -1;  while (isspace(p[i]) && p[i] != '\0')  {  i++;  }  while (isspace(p[j]) && j>0 )  {  j--;  }  ncount = j - i + 1;  //  strncpy(buf2, p+i, ncount);  buf2[ncount] = '\0';  return ret;  }  //求去掉空格  //int trimSpaceStr2(char \*p, unsigned char \*buf2, int \*buf2len)  //不要轻易去改变指针输入特性中in内存块的内存。。。。  int trimSpaceStr2\_notgood( char \*p)  {  int ret = 0;  int ncount = 0;  int i, j;  i = 0;  j = strlen(p) -1;  while (isspace(p[i]) && p[i] != '\0')  {  i++;  }  while (isspace(p[j]) && j>0 )  {  j--;  }  ncount = j - i + 1;  //  strncpy(p, p+i, ncount);  p[ncount] = '\0';  return ret;  } |
| 字符串反转模型 |
| void main51()  {  char p[] = "abcde";  char c ;  char \*p1 = p;  char \*p2 = p + strlen(p) -1;  while (p1 < p2)  {  c = \*p1;  \*p1 = \*p2;  \*p2 = c;  ++p1;  --p2;  }  printf("p:%s \n", p);  system("pause");  } |
|  |

## 两个辅助指针变量挖字符串

|  |
| --- |
| int getKeybyValue(char \*pKeyValude, char \*pKey, char \*pValude)  {  char rv = 0;  char \*p = NULL;  if (pKeyValude==NULL )  {  rv = -1;  printf("func getKeybyValue() err:%d pKeyValude \n", rv);  return rv;  }  if ( pKey==NULL )  {  rv = -1;  printf("func getKeybyValue() err:%d pKey=NULL \n", rv);  return rv;  }  if ( pValude==NULL )  {  rv = -1;  printf("func getKeybyValue() err:%d pValude \n", rv);  return rv;  }  //1 在pKeyValude中查找是否有关键字pKey  p = strstr(pKeyValude, pKey);  if (p == NULL)  {  rv = -1;  printf("func getKeybyValue() err:%d 查找没有关键字pKey \n", rv);  return rv;  }  p = p + strlen(pKey); //为下一次检索做准备  //2 有没有=  p = strstr(p, "=");  if (p == NULL)  {  rv = -2;  printf("func getKeybyValue() err:%d 查找没有= \n", rv);  return rv;  }  p = p + 1; //为下一次提取valude做准备    //3 提取按照要求的valude  rv = trimSpaceStr03(p, pValude);  if (rv != 0)  {  printf("func trimSpaceStr03() err:%d \n", rv);  return rv;  }  return rv;  } |
|  |

## 项目开发易错模型建立

|  |
| --- |
| 建立一个思想：是主调函数分配内存，还是被调用函数分配内存； |
| //不要相信，主调函数给你传的内存空间，你可以写。。。。。。一级指针你懂了。  但是二级指针，你就不一定懂。。。抛出。。。。。。。。。 |
|  |
| 越界 语法级别的越界 |
| char buf[3] = "abc"; |
|  |
| 不断修改指针变量的值 |
|  |
| 临时str3内存空间 |
| char \*str\_cnct(char \*x, char\* y) /\*简化算法\*/  {  char str3[80];  char \*z=str3; /\*指针z指向数组str3\*/  while(\*z++=\*x++);  z--; /\*去掉串尾结束标志\*/  while(\*z++=\*y++);  z=str3; /\*将str3地址赋给指针变量z\*/  return(z);  }  2、经验要学习  while(\*z++=\*x++);  z--; /\*去掉串尾结束标志\*/ |

const专题讲座

|  |
| --- |
| Const好处  //合理的利用const，  //1指针做函数参数，可以有效的提高代码可读性，减少bug；  //2清楚的分清参数的输入和输出特性 |
| 结论：  //指针变量和它所指向的内存空间变量，是两个不同的概念。。。。。。  //看const 是放在\*的左边还是右边 看const是修饰指针变量，还是修饰所指向的内存空变量 |
| int main()  {  const int a; //  int const b;  const char \*c;  char \* const d;  const char \* const e ;  return 0;  }  Int func1(const )  初级理解：const是定义常量==》const意味着只读 |
| 含义：  //第一个第二个意思一样 代表一个常整形数  //第三个 c是一个指向常整形数的指针(所指向的内存数据不能被修改，但是本身可以修改)  //第四个 d 常指针（指针变量不能被修改，但是它所指向内存空间可以被修改）  //第五个 e一个指向常整形的常指针（指针和它所指向的内存空间，均不能被修改） |
|  |

# 04二级指针输入模型专题讲座

## 01二级指针输入模型概念

|  |
| --- |
|  |
|  |

## 02多维数组名的本质

|  |
| --- |
| Char myArray[10][30]指针数组的一个指针.  myArray是一个指针变量 ，是一个常量。。。是一个常量指针 |
|  |

## 03多维数组做函数参数退化问题

|  |
| --- |
| void f(int a[5]) ====》void f(int a[]); ===》 void f(int\* a);  void g(int a[3][5])====》 void g(int a[][5]); ====》 void g(int (\*a)[5]); |
| 技术推演过程 \*(\*(a+1) +j ) a[i][j] |
|  |

## 04第1种和第3中二级指针做函数参数退化问题

|  |
| --- |
| Chsr \* p[3] = {“aaaa”, “bbb”,”cccc”};  Int printArray(char \*p[3])==🡺Int printArray(char \*p[])==🡺Int printArray(char \*\*p) |
|  |

## 05数组类型、数组指针类型、数组指针类型变量

|  |
| --- |
| void main()  {  //03、数组类型、数组指针类型、数组指针类型变量  typedef int MyTypeArray[5];  MyTypeArray a; //int a[5];  int intArray[3][5];  {  typedef int (\*MyPTypeArray)[5];  MyPTypeArray myArrayPoint ;  myArrayPoint = &a;  (\*myArrayPoint)[0] = 1; //通过一个数组指针变量去操作数组内存  }  {  int (\*myArrayVar)[5]; //告诉编译给我开辟4个字节的内存‘  myArrayVar = &a;  (\*myArrayVar)[1] = 2;    }  {  int (\*myArrayVar2)[5]; //告诉编译给我开辟4个字节的内存‘  myArrayVar2 = intArray; //  }  } |

## 06多维数组做函数参数退化原因大剖析

|  |
| --- |
| 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。所以说是真。。。。。 |
| /证明一下多维数组的线性存储  //线性打印  //多维数组做函数参数，一般情况下，只能表达到二维，  //如果是三维内存（我们程序员起的名字），已经没有意义。  //一般情况：1级指针代表1维，二级指针代表二维。。。  //如果有超过char \*\*\*级及3级以上的指针，则不代表几维的内存。。。  void printfAARRR(char \*\*\*ddd);  void printfAARRR(char \*\*\*\*\*\*\*\*\*dddd);  void printfArray411(int \*array, int num)  {  int i = 0;  for (i=0; i<num ; i++)  {  printf("%d ", array[i]);  }  }  void printfArray412(int (\*array)[5], int num)  {  return ;  }  void printfArrr333(int c[3][4][5])  {  return ;  }  void main()  {  int a[3][5];  int c[3][4][5];  int i , j = 0;  int tmp = 0;  for (i=0; i<3; i++)  {  for (j=0; j<5; j++)  {  a[i][j] = tmp ++;  }  }  printfArray411((int \*)a, 15);  system("pause");  } |

|  |
| --- |
| 1. C语言中只会以机械式的值拷贝的方式传递参数（实参把值传给形参） |
| int fun(char a[20], size\_t b) {    printf("%d\t%d",b,sizeof(a)); } |
| 原因1：高效 |
| 原因2： C语言处理a[n]的时候，它没有办法知道n是几，它只知道&n[0]是多少，它的值作为参数传递进去了 虽然c语言可以做到直接int fun(char a[20])，然后函数能得到20这个数字，但是，C没有这么做。 |
|  |
| 2、二维数组参数同样存在退化的问题 |
| 二维数组可以看做是一维数组  二维数组中的每个元素是一维数组  二维数组参数中第一维的参数可以省略  void f(int a[5]) ====》void f(int a[]); ===》 void f(int\* a);  void g(int a[3][3])====》 void g(int a[][3]); ====》 void g(int (\*a)[3]); |
| 3、等价关系 |
| 数组参数 等效的指针参数    一维数组 char a[30] 指针 char\*  指针数组 char \*a[30] 指针的指针 char \*\*a  二维数组 char a[10][30] 数组的指针 char(\*a)[30] |

## 07二级指针三种内存模型建立

|  |
| --- |
|  |
|  |

//C:概念不清晰是产生bug的根源

//C即使概念不清晰，训练不到位，也是产生bug的根源===》避免眼高手低、训练到极致

//C：不能深入理解C各种语法现象，是阻碍你成为高手的主要原因。

## 08第三种内存模型强化

|  |
| --- |
|  |
| char \*\*getMem(int count)  {  int i = 0;  char \*\*tmp = (char \*\*)malloc(count\*sizeof(char \*));  for (i=0; i<count; i++)  {  tmp[i] = (char \*)malloc(100);  }  return tmp;  }  void sortArray(char \*\*myArray, int count)  {  int i = 0, j = 0;  char \*tmp;  for (i=0; i<count; i++)  {  for (j=i+1; j<count; j++)  {  if (strcmp(myArray[i], myArray[j]))  {  tmp = myArray[i]; //这个地方交换的是指针变量  myArray[i] = myArray[j];  myArray[j] = tmp;  }  }  }  }  void sortArray02(char \*\*myArray, int count)  {  int i = 0, j = 0;  char tmp[200];  for (i=0; i<count; i++)  {  for (j=i+1; j<count; j++)  {  if (strcmp(myArray[i], myArray[j]) > 0)  {  strcpy(tmp, myArray[i]);  strcpy(myArray[i], myArray[j]);  strcpy(myArray[j], tmp); //交换是buf的内容  }  }  }  }  void printfArray(char \*\*myArray, int count)  {  int i = 0, j = 0;  for (i=0; i<count; i++)  {  printf("%s \n", myArray[i]);  }  }  void main()  {  char \*\*pArray = NULL;  pArray = getMem(3);  strcpy(pArray[0], "bbbbb");  strcpy(pArray[1], "aaaa");  strcpy(pArray[2], "cccc");  printf("排序之前\n");  printfArray(pArray ,3);  //sortArray(pArray, 3);  sortArray02(pArray, 3);  printf("排序之后\n");  printfArray(pArray ,3);  system("pause");  } |

## 09第三种内存模型结束标志

|  |
| --- |
| char \*\*getMem(int count)  {  int i = 0;  char \*\*tmp = (char \*\*)malloc((count+1)\*sizeof(char \*) );  for (i=0; i<count; i++)  {  tmp[i] = (char \*)malloc(100);  }    tmp[count] = '\0'; //转义字符的0  tmp[count] = 0; //转义字符的0  tmp[count] = NULL; //转义字符的0  return tmp;  } |
|  |

## 10野指针产生原因及解决方案

### 基础知识

|  |
| --- |
| //野指针产生问题分析  //指针变量和它所指内存空间变量是两个不同的概念  //解决野指针的方案  //1定义指针时 把指针变量赋值成null  //2 释放内存时，先判断指针变量是否为null  //3 释放内存完毕后，把指针变量重新赋值成null |

### 野指针和1级指针做函数参数在一起

|  |
| --- |
| #include "stdio.h"  #include "stdlib.h"  #include "string.h"  //野指针产生问题分析  //指针变量和它所指内存空间变量是两个不同的概念  //解决野指针的方案  //1定义指针时 把指针变量赋值成null  //2 释放内存时，先判断指针变量是否为null  //3 释放内存完毕后，把指针变量重新赋值成null  //  void main22()  {  char \*p = NULL;  p = (char \*)malloc(100); //char p[100];  strcpy(p, "abcdefg");  //做业务  //此处省略5000字。。。。。  if (p != NULL)  {  free(p);  p = NULL;  }  //做业务  //此处省略5000字。。。。。  if (p != NULL)  {  free(p);  }  system("pause");  }  char \*getMem2(int count)  {  char \*tmp = NULL;  tmp = (char \*)malloc(100\*sizeof(char)); //char tmp[100];  return tmp;  }  //实参和形参是两个不同的概念  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  //直接修改实参没戏。。。。。。。 实参和形参是两个不同的概念  //return tmp;  }  void getMem4(int count, char \*\*p /\*out\*/)  {  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  //直接修改实参没戏。。。。。。。 实参和形参是两个不同的概念  //间接的修改实参  //\*(实参的地址) =  \*p = tmp;  //return tmp;  }  //函数调用的时候，这个场景修改不实参  int FreeMem2(char \*p)  {  if (p ==NULL)  {  return -1;  }  if (p != NULL)  {  free(p);  p = NULL; //想把实参给改掉，你能修改吗？ 修改不了实参。。。。。  }  return 0;  }  void main51()  {  char \*myp = NULL;  myp = getMem2(100);  //getMem3(100, myp);  //getMem4(100, &myp);  //做业务操作  //此 50000  FreeMem2(myp);  FreeMem2(myp);  } |
|  |

# 05结构体专题讲座

## 01、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 |
|  |
|  |

## 02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;  } |

## 03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  } |

## 04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);  } |
|  |

## 结构体的高级话题

|  |
| --- |
| 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");  } |

# 06文件专题讲座

文件读写api的熟悉

|  |
| --- |
|  |
| char \*fname = "c:\\1.txt";  char \*fname2 = "c:/a1.txt"; //统一的用45度斜杠 |
| fgetc fputc 按照字符读写文件  fputs fgets 按照行读写文件 （读写配置文件）  fread fwirte 按照块读写文件 （大数据块迁移） |
|  |
| void main04()  {  int i = 0;  FILE \*fp = NULL;  char buf[100];  char \*p = NULL;  char \*fname = "c:\\1.txt";  char \*fname2 = "c:/a1.txt"; //统一的用45度斜杠  fp = fopen(fname2, "r"); //不管文件是否存在，新建文件  if (NULL == fp)  {  printf("func fopen（） err: \n");  }  while (!feof(fp))  {  //\_cdecl fgets(\_Out\_z\_cap\_(\_MaxCount) char \* \_Buf, \_In\_ int \_MaxCount, \_Inout\_ FILE \* \_File);  p = fgets(buf, 100, fp);  if (p == NULL)  {  printf("func fgets() .....\n");  return ;  }  printf("%s \n", buf);  printf("%s \n", p);  }    if (fp != NULL)  {  fclose(fp);  }  } |
| 项目开发中参考fgets函数的实现方法  fgets(buf, bufMaxLen, fp);  对fgets函数来说，n必须是个正整数，表示从文件按中读出的字符数不超过n-1，存储到字符数组str中，并在末尾加上结束标志’\0’，换言之，n代表了字符数组的长度，即sizeof(str)。如果读取过程中遇到换行符或文件结束标志，读取操作结束。若正常读取，返回指向str代表字符串的指针，否则，返回NULL（空指针）。 |
|  |
|  |
| 文件控制 |
| fp = fopen(pFileName, "r+");  if (fp == NULL)  {  rv = -2;  printf("fopen() err. \n");  //goto End;  }  if (fp == NULL)  {  fp = fopen(pFileName, "w+t");  if (fp == NULL)  {  rv = -3;  printf("fopen() err. \n");  goto End;  }  }    fseek(fp, 0L, SEEK\_END); //把文件指针从0位置开始，移动到文件末尾  //获取文件长度;  length = ftell(fp);  fseek(fp, 0L, SEEK\_SET); |

配置文件读写库的设计与实现

|  |
| --- |
|  |
| // cfg\_op.h  #ifndef \_INC\_CFG\_OP\_H  #define \_INC\_CFG\_OP\_H  #ifdef \_\_cplusplus  extern "C" {  #endif  int GetCfgItem(char \*pFileName /\*in\*/, char \*pKey /\*in\*/, char \* pValue/\*in out\*/, int \* pValueLen /\*out\*/);  int WriteCfgItem(char \*pFileName /\*in\*/, char \*pItemName /\*in\*/, char \*pItemValue/\*in\*/, int itemValueLen /\*in\*/);  //int CfgItem\_Init(void \*pHandle, int iType);  //int GetCfgItem(void \*pHandle /\*in\*/, char \*pKey /\*in\*/, char \* pValue/\*in out\*/, int \* pValueLen /\*out\*/);  //int WriteCfgItem(void \*pFileName /\*in\*/, char \*pItemName /\*in\*/, char \*pItemValue/\*in\*/, int itemValueLen /\*in\*/);  //int CfgItem\_Destory(void \*pHandle);  #ifdef \_\_cplusplus  }  #endif  #endif |

大数据文件加密解密设计与实现

# 指针专题讲座

## 铁律1：指针是一种数据类型

1. 指针也是一种变量，占有内存空间，用来保存内存地址

测试指针变量占有内存空间大小

2）\*p操作内存

在指针声明时，\*号表示所声明的变量为指针

在指针使用时，\*号表示 操作 指针所指向的内存空间中的值

\*p相当于通过地址(p变量的值)找到一块内存；然后操作内存

\*p放在等号的左边赋值（给内存赋值）

\*p放在等号的右边取值（从内存获取值）

3）指针变量和它指向的内存块是两个不同的概念

//含义1 给p赋值p=0x1111; 只会改变指针变量值，不会改变所指的内容；p = p +1; //p++

//含义2 给\*p赋值\*p='a'; 不会改变指针变量的值，只会改变所指的内存块的值

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

//含义4 =左边char \*p

//含义5 保证所指的内存块能修改

4）指针是一种数据类型，是指它指向的内存空间的数据类型

含义1：指针步长（p++），根据所致内存空间的数据类型来确定

p++=🡺(unsigned char )p+sizeof(a);

结论：指针的步长，根据所指内存空间类型来定。

注意： 建立指针指向谁，就把把谁的地址赋值给指针。图和代码和二为一。

不断的给指针变量赋值，就是不断的改变指针变量（和所指向内存空间没有任何关系）。

## 铁律2：通过\*p/\*p++ 来改变变量的值是指针存在的最大意义

1）两码事：指针变量和它指向的内存块变量

2）条件反射：指针指向某个变量，就是把某个变量地址否给指针

3）\*p间接赋值成立条件：3个条件

a)2个变量（通常一个实参，一个形参）

b) 建立关系，实参取地址赋给形参指针

c) \*p形参去间接修改实参的值

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

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

//改变0级指针（int iNum = 1）的值有2种方式

//改变1级指针（eg char \*p = 0x1111 ）的值，有2种方式

//改变2级指针的（eg char \*\*pp1 = 0x1111 ）的值，有2种方式

//函数调用时，形参传给实参，用实参取地址，传给形参，在被调用函数里面用\*p，来改变实参，把运算结果传出来。

//指针作为函数参数的精髓。

## 铁律3：理解指针必须和内存四区概念相结合

1. 主调函数 被调函数
   1. 主调函数可把堆区、栈区、全局数据内存地址传给被调用函数
   2. 被调用函数只能返回堆区、全局数据
2. 内存分配方式
   1. 指针做函数参数，是有输入和输出特性的。

## 铁律4：应用指针必须和函数调用相结合（指针做函数参数）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 编号 | 指针函数参数  内存分配方式（级别+堆栈） | | 主调函数  实参 | 被调函数  形参 | | | | 备注 |
| 01 | 1级指针  （做输入） | 堆 | 分配 | 使用 | | | | 一般应用禁用 |
| 栈 | 分配 | 使用 | | | | 常用 |
| Int showbuf(char \*p);  int showArray(int \*array, int iNum) | | | | | | |
| 02 | 1级指针  （做输出） | 栈 | 使用 | 结果传出 | | | | 常用 |
| int geLen(char \*pFileName, int \*pfileLen); | | | | | | |
| 03 | 2级指针  （做输入） | 堆 | 分配 | 使用 | | | | 一般应用禁用 |
| 栈 | 分配 | 使用 | | | | 常用 |
| int main(int arc ,char \*arg[]); 指针数组  int shouMatrix(int [3][4], int iLine);二维字符串数组 | | | | | | |
| 04 | 2级指针  （做输出） | 堆 | 使用 | 分配 | | 常用，但不建议用，转化成02 | | |
| int getData(char \*\*data, int \*dataLen);  Int getData\_Free(void \*data);  Int getData\_Free(void \*\*data); //避免野指针 | | | | | | |
| 05 | 3级指针  （做输出） | 堆 | 使用 | | 分配 | | 不常用 | |
| int getFileAllLine(char \*\*\*content, int \*pLine);  int getFileAllLine\_Free(char \*\*\*content, int \*pLine); | | | | | | |

指针做函数参数，问题的实质不是指针，而是看内存块，内存块是1维、2维。

1. 如果基础类int变量，不需要用指针；
2. 若内存块是1维、2维。

## 铁律5：一级指针典型用法（指针做函数参数）

一级指针做输入

int showbuf(char \*p)

int showArray(int \*array, int iNum)

一级指针做输出

int geLen(char \*pFileName, int \*pfileLen);

理解

主调函数还是被调用函数分配内存

被调用函数是在heap/stack上分配内存

## 铁律6：二级指针典型用法（指针做函数参数）

二级指针做输入

int main(int arc ,char \*arg[]); 字符串数组

int shouMatrix(int [3][4], int iLine);

二级指针做输出

int Demo64\_GetTeacher(Teacher \*\*ppTeacher);

int Demo65\_GetTeacher\_Free(Teacher \*\*ppTeacher);

int getData(char \*\*data, int \*dataLen);

Int getData\_Free(void \*data);

Int getData\_Free2(void \*\*data); //避免野指针

理解

主调函数还是被调用函数分配内存

被调用函数是在heap/stack上分配内存

## 铁律7： 三级指针输出典型用法

三级指针做输出

int getFileAllLine(char \*\*\*content, int \*pLine);

int getFileAllLine\_Free(char \*\*\*content, int \*pLine);

理解

主调函数还是被调用函数分配内存

被调用函数是在heap/stack上分配内存

## 铁律8：杂项，指针用法几点扩充

1）野指针 2种free形式

int getData(char \*\*data, int \*dataLen);

int getData\_Free(void \*data);

int getData\_Free2(void \*\*data);

2）2次调用

主调函数第一次调用被调用函数求长度；根据长度，分配内存，调用被调用函数。

3）返回值char \*/int/char \*\*

4）C程序书写结构

商业软件，每一个出错的地方都要有日志，日志级别

## 铁律9：一般应用禁用malloc/new

## 铁律10：C函数指针是C++至高无上的荣耀