- 指针
  - 指针的概念
    - 变量的地址与指针变量
      - 变量的地址
      - 指针变量
      - 指针运算符
    - 指针变量的声明
    - 指针的赋值和移动操作
      - 指针的赋值
      - 指针的移动
    - 悬挂指针与NULL指针
      - 悬挂指针
      - NULL指针
  - 指针参数
    - 传值调用和传址调用
    - 返回多个值的函数
  - 指针和一维数组
    - 一维数组元素的指针表示
      - 指向数组元素的指针
      - 指针的关系运算
      - 指针的自减运算
      - 单目*和++的组合
    - 一维数组参数的指针表示
      - 用指向数组元素的指针作函数参数
      - 指向常量的指针作为函数参数
      - 常量指针和指针常量
  - 指针和字符串
    - 字符串的指针表示
    - 字符串作函数参数
  - 指针数组
    - 指针数组的概念
      - 指针数组的声明
      - 指针数组的初始化
    - 用指针变量表示字符串数组
      - 字符指针数组的初始化
      - 指针数组与二维数组的比较
      - 动态分配字符串数组
      - 指针数组作函数参数

指针

制作类型是C语言最具特色的数据类型,指针变量是一种用来存放地址的特殊变量

指针的概念

指针就是地址,指针变量就是专门存放地址的变量

变量的地址与指针变量

变量的地址

程序中的任何变量都占据一定数量的**以字节为单位的内存单元**,所需内存单元的字节数由变量的类型决定.内存的每一个字节都有一个编号,内存字节的编号称为**地址**.
变量在内存存储中的起始地址称为变量的地址.程序编译后,就将变量名转换为了变量的地址.变量名与内存地址之间的关联是**编译器**实现的,计算机通过地址来存取变量值.

变量的地址是分配给这个变量的内存单元起始位置编号,变量的值是该内存位置所存储的数值.
地址是一类比较特殊的数据,代表某个变量所占内存的编号,相当于一个指示器.因此,**一个变量的地址称为该变量的指针**.
指针是C语言的一种**数据类型**,和其他数据类型一样有常量和变量之分.
若声明 `int a,b[10];` ,则 `&a` 是变量a的地址,其类型为指针,是一个**指针常量**; `b` 或 `&b` 或 `&b[0]` 类型也是指针,是一个**指针常量**.因为系统分配给变量a和数组b的内存位置是不变的,**不能被程序修改的地址值称为指针常量**.

指针变量

可以将变量a的地址存放在另一个变量p中,赋值语句 `p=&a;` 用于将a的地址存入变量p中,称为"p指向变量a",或"p所指的对象是a",或"**p是a的指针**".
注意**变量p的值是&a,属于指针类型**,而变量p本身也有地址&p.
专门用来存放地址值的变量称为指针变量,p就是一个指针变量,只能将地址值赋值给他.指针变量是一类特殊的变量,它是用来指向另一个变量的,引入指针的目的是通过指针间接访问它所指向变量.如果p指向a,通过**访问p**能够知道a的地址,从而找到a的内存单元,取到其值.

指针运算符

- 单目运算符*是**指针运算符(间接访问运算符)**,其操作数类型必须是指针,用来访问指针所指向的对象.如果指针变量指向a,则语句 `*p=200;` 等价于 `a=200;` ,将200赋值给所指的对象.
- 利用**变量名**实现对变量的访问称为**直接访问**,通过**变量的指针**实现对变量的访问称为**间接访问**.
- p是变量,其值可以更改,可以改变为存放其他变量的地址.例如 `p=&b[0];` ,指针p指向变量b[0],*p就是间接访问b[0],执行语句中出现的b[0]都可以用*p代替.
- 如果p不是指针类型,但是p存储的是地址,那么使用*p来访问该地址通常是不合法的,并且可能会导致编译时错误或运行时错误.因为解引用操作*通常是用于指针类型的,它告诉编译器去访问指针所指向的内存地址,而非简单地将地址视为数据本身.

指针变量的声明

指针变量是一种特殊类型的变量,相比于其他变量多了一个记号*.

- 例如, `int *p;` 声明语句中变量名前的"*"是**指针说明符**,说明该变量是指针变量,int说明指针p所指的对象类型是int.称"**p是指向整型数据的指针变量**"或"p是整型指针变量",表达式*p的类型就是int,通过p间接访问所指地址处连续的4字节单元内容.编译器会根据**指针的类型**来确定应该读取或写入多少个字节的数据.
- 若有声明 `char *pc;` ,则pc是指向字符型数据的指针变量.表达式*pc的类型是char,访问所指地址处1字节单元的内容.
- 声明指针的一般形式是: `类型区分符 *标识符1,*标识符2,...,*标识符n;` 标识符是**指针变量名**,指针所指**对象的类型**由类型区分符决定.

- 指针变量可以在声明的时候对其进行初始化,但必须用变量的地址进行初始化,初始化后的指针指向该变量.
- 例如 `int x,*p=&x;` 该声明语句是指**给p赋值,而不是给*p赋值**,使指针p指向变量x,出现在声明语句中的星号*是指针说明符,仅仅标记其后变量是指针变量,而不进行间接访问运算,初始化是针对指针变量的.上面的声明语句等价于 `int x,*p; p=&x;`
- 程序中的*有三种含义:
  - 双目的乘法运算符;
  - 单目的指针**间接访问运算符**;
  - 在**声明语句**中是**指针说明符**,此时的*读作"指针".

指针的赋值和移动操作

指针的赋值

- 可以将一个指针常量赋值给一个指针变量,也可以将一个指针变量赋值给另一个指针变量,**如果两个指针类型相同,则直接复制**(指针类型就是指针所指变量的类型).
- 例如 `int x,*p1,*p2; p1 = &x ; p2 = p1;` ,&x的类型是**int *(整型指针)**,p1,p2的类型也是int *,直接将整型指针常量&x赋值给整型指针变量p1.由于p1是变量,可以通过直接访问方式将p1的值赋给p2,二者均指向变量x.
- **如果两个指针类型不同,一般要通过类型强制符转换后再赋值**.
  指针的类型转换规则主要涉及两个方面:隐式类型转换和显式类型转换.
  - **隐式类型转换**:在一些情况下,指针类型之间可以隐式地进行转换,通常在类型之间存在一定的兼容性时才会发生.例如,在C语言中,可以将指向派生类型的指针转换为指向基类型的指针,而不需要显式地进行转换.但是反之则通常不行,因为基类型的指针可能无法正确地访问派生类型的数据.
  - **显式类型转换**:有时候,需要进行显式的指针类型转换,这可以通过在指针前加上括号并提供目标类型来实现.例如,在C语言中,可以使用强制类型转换来将一个指针从一种类型转换为另一种类型.这种转换在某些情况下是必要的,但同时也需要谨慎使用,因为它可能导致类型不匹配或者数据错位的问题.

  指针间接访问的结果取决于指针类型,因此指针变量的声明必须指定指针类型,这样才能利用指针正确的进行间接访问运算.在编译器中,对取地址运算符&的使用是基于该变量的类型的.编译器会根据变量的类型来生成对应的指针类型.所以,虽然&操作符本身不包含类型信息,但在编译时,它的行为会受到变量类型的影响,以生成正确类型的指针.

E.G.

```
short x=0x1234, *p1;
char *p2;
p1 = &x;/* 同类型指针直接复制,p1指向x */
p2 = (char *)&x;
/* 不同类型指针赋值要强制转换,p2指向x的首字节但允许访问字节数发生变化 */
/* 事实上,可以也通过隐式类型转换,直接写作 p2=&x;或者 char *p2=&x */
printf("%#x %#x",*p1,*p2);/* 输出 0x1234 0x34 */
```

注意

数据的**机器码在内存中的存储顺序是从高字节(内存编码更高)开始**,由高到低,而数据**首字节对应内存编码最低的字节**.即高字节数据放在高地址,低字节数据放在低地址对于x=0x1234,12存储在最高字节,而34存储在最低字节,34对应的字节编码是变量x的地址.强制类型转换时,指针指向的地址不变,但**间接访问的字节数变化**,高字节的12被截去.
此外,**变量的内存寻址也是从高字节到低字节的,和机器码寻址一样.但数组中每一个元素对应地址的排序又是从低字节到高字节的**.

指针的移动

指针的移动是通过将指针加一个整数实现指针的后移(地址增大的方向),或者通过指针减一个整数实现指针的前移(地址减少的方向).假设p是指针,n是整数,则表达式(p+n或p-n)的类型和p一样,仍是指针,它指向**p当前位置前后的第n个元素(而不是字节)**.我们可以通过操作指针,利用间接访问运算符*,实现间接操作指针所指向的字节对象.

```
/* 将short数的高字节和低字节交换 */
    short a=0x5678;
    char t,*p=(char *)&a;/* p指向a的低字节 */
    printf("Before swap a=%#hx\n",a);
    t=*p; *p=*(p+1); *(p+1)=t;/* 交换高低字节,通过操作指针间接操作指针指向的字节 */
    printf("After swap a= %#hx\n",a);
    return 0;
```

这个程序中通过 `(char *)` 对&a进行了指针强制类型转换,但a本身的类型并没有改变.

```
int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr; // 指向数组的第一个元素
    // 访问数组中的元素
    printf("数组元素: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr); // 解引用指针，访问当前指向的元素
        ptr++; // 移动指针到下一个元素
    }
    printf("\n");
    return 0;
```

在这个代码中,使用 `ptr++` 而不是 `ptr += 4` 是因为 `ptr` 是一个指向 `int` 类型的指针.在C语言中,**指针的自加操作会根据指针所指向的类型来计算偏移量**.
假设 `ptr` 指向一个 `int` 类型的数组元素,那么 `ptr++` 会使指针向后移动一个 `int` 大小的位置,这样它会指向下一个数组元素.如果使用 `ptr += 4` ,则会使指针向后移动4个 `int` 大小的位置,这将导致它跳过了4个数组元素,而不是指向下一个数组元素.
因此,为了正确地遍历整个数组,我们应该使用 `ptr++` 来使指针逐个移动到下一个数组元素的位置.

其它代码见D:\C_code\C\Pointer\pointer_2\pointer_2(取出字节的高位与低位转换为16进制).c

悬挂指针与NULL指针

悬挂指针

如果一个指针没有被设置为指向一个已知的对象，则这样的指针称为悬挂指针,例如下列代码:

```
char *p1;/* 字符指针变量,未初始化,无所指 */
long *p2;/* 长整型变量,未初始化,无所指 */
scanf("%s",p1);
*p2=10;
```

当创建一个指针时,系统只分配了用来**存储指针本身的内存空间**,不会分配用来**存储数据的内存空间**. `char *p1;` 创建了一个名为p1的字符指针变量, `scanf("%s",p1);` 从键盘读入一个字符串存放到指针p1指向的内存空间.但p1无所指.
指针变量和其他变量一样,声明后没有被初始化,如果是自动变量,它的值是**随机的**,一般认为指针处于无所指的状态,是一个**悬挂指针(野指针)**.
如果仍要执行,运气好的话,p1和p2的初始值会指向一个**非法地点(用户没有访问权限)**,称为内存错误或者段违规.更严重的是指针初始值指向一个合法地点,该位置上的数据被覆盖.

NULL指针

C语言允许指针**具有0值**,通常用符号常量NULL表示,称为**空指针或NULL指针**,表示**不指向任何对象**.NULL在头文件stdio.h和stdlib.h均有定义.
NULL可以赋给**任何类型的指针变量**,变量声明时,如果没有确切的地址可以赋值,应当为指针变量赋一个NULL值,实际上就是赋值为0.NULL是一个特殊的指针值,他表示指针目前88未指向任何对象处于不能用的状态**.程序任何时候不能向NULL指针的对象赋值,因为内存地址为0开始的一块区域作为**中断向量区**,不能随意访问.
空指针有确定的值0,可以确保不指向任何对象;悬挂指针不为确定值,可以指向任何地方.

指针参数

传值调用和传址调用

执行函数调用时,系统**为每个形参分配了存储空间**,将实参的值传递给形参单元,形参是实参的**副本**,因此在被调用函数中对形参的修改不会改变实参变量的值.例如利用swap函数交换两个变量的值:

```
void swap(int x,int y)
{
    int t;
    t=x;x=y;y=t;
}
```

函数调用 `swap(a,b);` 不能实现交换a和b的值,因为实参和形参占据占据**不同的存储单元**,实参仅仅将其值传递给形参,在swap函数中交换的形参x和y只是a和b的副本,而不是a和b本身,a和b不受影响.
为了交换调用函数中的变量a和b,需要将a和b的**地址(指针)作为参数**传递给调用函数的形参,使形参指向a,b,再被调用函数中通过指针**间接访问调用函数中的a和b本身**.因此swap函数的调用应该写为 `swap(&a,&b);` swap函数定义为:

```
void swap(int *p1,int *p2)
{
    int t;
    t=*p1;*p1=*p2;*p2=t;
}
```

swap的形参都是整型指针,调用时传递的是地址,通过间接访问运算交换实参.
*p1和*p2既是swap的输入参数,又是输出参数.**通过指针参数的传址调用可以改变调用函数中变量的值**.实际上,C语言函数调用一直是**传值**的.只不过传递的值可以是变量值,也可以是变量的地址值.当使用传址方式时,形参要说明为指针,实参可以是指针常量,也可以是指向某个对象的指针变量.

返回多个值的函数

return语句只能从函数中返回一个值,**指针参数可以作输出参数将值传会调用函数,从而解决函数返回多值问题**.

```
void MonthDay(int year,int yearDay,int *pMonth,int *pDay)
{
    static int dayTab[][13]={
        {0,31,28,31,30,31,30,31,31,30,31,30,31},
        {0,31,29,31,30,31,30,31,31,30,31,30,31}
    };
    int i,leap;

    leap=(((year%4==0)&&(year%100!=0))||(year%400==0));
    for(i=1;yearDay>dayTab[leap][i];i++)
        yearDay-=dayTab[leap][i];
    *pMonth=i;
    *pDay=yearDay;
}
```

指针和一维数组

数组名被隐式地转换为指向数组第0个元素的指针,可以用指针代替下标引用数组元素,由于下标运算符[]是系统的**预定义函数**,下标操作实际上涉及对系统预定义函数的调用.因此指针操作比下标操作要快.

一维数组元素的指针表示

指向数组元素的指针

- 声明 `int x[6];` 定义了一个有6个元素的整型数组x,数组x存放在一片连续的内存区域中,该区域占6*sizeof(int)字节.为了用指针变量表示x的元素,先声明一个与x同类型的指针变量: `int *p;` ,并由赋值语句 `p=&x[0];` 或 `p=x;` 使p指向x的第0个元素.
- 指针变量p也占据内存,在64位系统上,指针变量通常占用8个字节(64位),因为地址空间的大小是64位.在指针变量p占据的内存中存放 x[0] 的地址,也可以更改为存放其他元素地址,数组名x固定指向 x[0] ,是一个指针常量.
- 当 p 指向 x[0] 时, (p+i) 指向 x[i] .因此 *p 等价于 x[0] , *(p+i) 等价于 x[i] .总之,数组元素的指针加1,等效于数组元素的下标加1.所以,若 p 指向 x[i] , ++p 使 p 指向 x[i+1] , *(++p) 等价于 x[++i].
- 由于数组名也是一个指针常量,同理有:x 指向 x[0], (x+i) 指向 x[i], (x+i) 等价于 &x[i], *(x+i) 等价于 x[i].

- 由此可见,引用数组元素有两种等价形式:**通过下标引用和通过指针引用**.注意指针变量和数组名是有区别的:p是变量,数组名x是常量.因此 `p=x+1;` , `p++;` , `&p` 等均是合法操作;但 `x=p;` , `x++;` 就不是合法操作.
- 可以用**指针变量带下标的形式引用数组元素**.若p指向x[2],则 p[0] 等价于 *p 等价于 x[2] 等价于 *(x+2) , p[i] 等价于 *(p+i) 等价于 x[2+i] 等价于 *(x+2+i).

指针的关系运算

两个指针可以进行比较关系运算,但只有当他们**指向同一数组**时,比较运算才有意义.实际应用时,指针的关系运算多用在循环条件控制中.

```
#define N 10
int main(void)
{
    int a[N], *p;
    printf("Please enter %d numbers\n",N);
    for(p=a;p<a+N;p++)
        scanf("%d",p);
    while(--p>=a)//for循环结束后,p指向a[10],为了逆序输出数组元素,先将p回退a[9],因此使用前置式自减--p
        printf("%d",*p);
    return 0;
}
```

指针的自减运算

指向同一数组的两个指针可以进行减法运算,两个指针相减的结果是**所指元素的下标之差**,即相隔元素个数,值是一个整数.

```
int strlen(char *s)
{
    char *p=s;
    while(*p!='\0')
        p++;
    return (p-s);/* p-s就是字符串长度 */
}
```

单目*和++的组合

++,--和*都位于**第二优先级,结合性是右结合**.
设 `int y,x[3]={1,3,5},*p=x;` ,则

- `++*p` 等价于 `++(*p)` ,p指向x[0],*p等价于x[0],然后对x[0]做++操作,x[0]的值由1变为2(事实上,x[i]是变量);
- `y=*++p` 等价于 `y=*(++p)` ,p先自增,使p指向x[1],再作间接访问运算,得到y=3;
- `y=*p++` 等价于 `y=*(p++)` ,由于是后置++,p还是指向x[0],从而x[0]被赋给y,表达式的值是1.然后p进行了自增,使p指向x[1].
- `y=(*p)++` 等价于 `y=x[0]++` ,后置++需要延迟计算,将x[0]的值赋给y后再执行x[0]的自增.表达式的值是1,x[0]变为2.

进栈与退栈

进栈(Push)和退栈(Pop)是栈(Stack)数据结构中的两个基本操作.
栈是一种具有后进先出(Last-In-First-Out,LIFO)特性的数据结构.在栈中,只能从栈顶进行插入(进栈)和删除(退栈)操作.这样保证了最后进入栈中的元素最先被移除.

- **进栈（Push）:** 进栈操作是向栈中放入新的元素.这个元素会被放置在栈顶,原有的栈顶元素(如果存在)将被向下移动一个位置.换句话说,进栈操作将一个新元素压入栈顶.
- **退栈（Pop）:** 退栈操作是从栈中移除元素.它会移除栈顶的元素,并返回该元素的值.栈的大小减小1,原来在栈顶下面的元素成为新的栈顶.

进栈和退栈操作是栈数据结构的核心功能,它们为实现栈的各种应用提供了基础支持.栈的进栈和退栈操作主要有以下应用:

1. **函数调用:** C语言使用栈来**管理函数调用**.当你调用一个函数时,函数的参数,局部变量以及函数返回地址等信息都会被压入栈中,当函数执行完毕后,这些信息就会被弹出栈,控制权回到调用函数的地方.
2. **表达式求值:** 在编译器中,栈常用于表达式求值.例如,中缀表达式转换为后缀表达式时,需要用到栈来存储操作符,以及在计算后缀表达式的值时,也需要用到栈来暂存中间结果.
3. **内存管理:** 栈在内存管理中也起到重要作用.在程序执行过程中,栈用于存储局部变量,函数参数和返回值等信息,以及保存函数调用的上下文,如返回地址,寄存器状态等.
4. **数据结构算法:** 在许多数据结构和算法中,栈也是一个重要的辅助工具.例如,深度优先搜索（Depth-First Search,DFS）,递归算法,括号匹配等都可以使用栈来实现.

`*p++=val;` 和 `val=*--p;` 在实际应用中是进栈和退栈操作的基本用法.p是栈指针,始终指向下一个可用单元val是准备进栈的值或从栈顶弹出的值.**进栈操作**:val先入栈,然后移动p指向下一个可用单元;**退栈操作**:移动p指向栈顶元素,然后取出栈顶元素.

一维数组参数的指针表示

用指向数组元素的指针作函数参数

用一维数组作函数参数时,形参说明为不指定大小的数组,实参用数组名.此时传递的其实是地址,被调用函数接受地址的形参是一个存放该地址的变量,就是**指针变量**.
冒泡排序函数 `void bubbleSort(int a[],int n){...}` 实际上可以等价地定义
为 `void bubbleSort(int *a,int n){...}` ,后者直接表明形参a是一个指针变量,数组名传递到函数时,函数可以用指针或者下标的方式引用数组元素.也可以向函数传递数组连续的一部分,例
如 `bubbleSort(a+2,int n);` 或 `bubbleSort(&a[2],int n);` .

```
void bubbleSort(int *a,int n)
{
    int t,i,*p;
    for(i = 0;i<n-1;i++){
    for(p = a;p<a+n-1-i;p++)
        if(*p>*(p+1)){
            t=*p;
            *p=*(p+1);
            *(p+1)=t;
        }
    }
}
```

指向常量的指针作为函数参数

如果函数仅仅使用原始数组数据,**并不需要修改这些数据**,那么在函数原型和函数定义的**形参声明中使用关键字const**.
使用const可以对实参数组提供保护,阻止函数修改传递过来的实参数组的数据.
如果函数要修改数组的内容,在声明数据时不使用const;如果函数不需要修改数组内容,在声明数组参数时最好使用const.

常量指针和指针常量

指针涉及两部分空间:**指针变量本身所占空间和指针所指对象所占的空间**,可以使用const将指针本身的内容限定为只读不能修改,也可以将指针指向的内容限定为只读不能修改.

1. **常量指针就是指向常量的指针**,表示不能**通过这个指针的间接访问**改变它所指向对象的值,但指针本身可以被修改.常量指针也可以指向某个变量,但值得注意的是:
   常量指针指向的内容是常量,编译器会在编译时进行检查,防止通过常量指针来修改这些内容.这种检查保证了代码的安全性和可靠性.

而当我们直接使用变量名来修改变量的值时,我们是**直接访问**存储在内存中的变量的值,并且没有通过指针间接访问.因此,编译器允许我们直接修改变量的值.

```
int a=5,b=6;
const int *p;//int const *p;
p=&a;/* 合法,可以修改指针的值 */
*p=10;/* 非法,不能通过指针间接访问修改变量a的值 */
a=8;/* 合法,a是变量,可以通过直接访问修改 */
```

2. **指针常量是指针类型的常量**,指针本身是常量,只读不能修改,但是指针指向的对象可以修改.

```
int a=5,b=6;
int * const p=&a;/* 指针是常量,必须初始化 */
*p=10;/* 合法 */
p=&b;/* 不合法 */
```

此外还可以声明 `const int * const p=&a;` .

指针和字符串

字符串的指针表示

程序中表示一个字符串有以下几种方法:

1. 声明一个字符指针变量和一个字符数组,字符串存放在字符数组里,并使指针变量指向字符数组,通过指针变量引用字符串或字符.

   ```
   char aStr[]="It is a string";
   char *pStr=aStr;
   ```

2. 声明和初始化一个**字符指针变量**,通过字符指针变量访问字符串或字符.

   ```
   char *pStr="It is a string";
   printf("%s,%c,%c\n",pStr,pStr[0],*(pStr+1));
   ```

   i. 字符串常量存储在一个**连续的无名存储区**,通过 `pStr="It is a string";` 将无名存储区的首地址赋给指针pStr,指针指向字符串的第0个字符.也就是说,指针变量保存的是字符串的首地址,而不是把字符串保存在pStr中.
   ii. 字符指针表示字符串的方式与字符数组很相似,都可以用%s输出整个字符串,也都有用*或[]引用单个字符.但本质上他们的声明形式有很多区别:

a. 首先他们的**存储结构**不同.前者aStr是字符数组,编译器会分配15B的存储空间,字符串放在这块存储区,数组名是存储区的起始地址,其类型为char *的常量;而后者pStr是字符指针变量,不管其类型是什么,编译器都分配8B内存(64位寻址长度),这个存储空间用来存地址.编译器会**另外给字符串分配15B空间**,并将该空间的首地址赋给pStr,这里pStr是变量.
b. 其次他们在内存中的**存储区域不同**.字符数组存储在**全局数据区(外部或全局数组)或栈区(自动数组)**,而指针表示的字符串存储在**常量区**,pStr存储在**全局数据区或栈区**.全局数据区和栈区的字符串有读取和写入的权限,而常量区的字符串只有读取权限,没有写入权限.
c. 内存权限不同的一个明显结果是:字符数组在定义后可以读取和修改每个字符;而对于指针表示的字符串,一旦被定义就只能读取不能修改.

```
char *str="ABC";
str="XYZ";/* 合法但不提倡 */
str[3]='D';/* Segment fault */
```

3. 和 `char *pStr="It is a string";` 类似还可以声明 `char (*pStr)[15]="It is a string";` ,它表示pStr是指向包含15个字符的字符数组的指针.其字符串的存储区域也在常量区.

字符串作函数参数

字符串作函数参数即字符数组作函数参数,它本质上是指向字符类型的指针作参数.函数定义的形参声明将形参声明为:char *或char[],函数调用时字符串首地址作实参.

```
/* 复制字符串函数 */
void myStrcpy(char *t,char *s)
{
    while((*t++=*s++)!='\0')//while(*t++=*s++)
    ;
}
```

该函数将指针s指向的字符串复制到指针t指向的存储单元,第一个串需要修改,但第二个串不需要修改,所以使用const.
若有 `char s1[20],s2[20],*ps1,*ps2;` ,则可以有以下调用形式:

```
myStrcpy(s1,"Sometimes I fantasize");
```

```
ps1=s1;ps2="Sometimes I fantasize";
myStrcpy(ps1,ps2);
```

```c
/* 字符串比较函数 */
int myStrcmp(const char*s1,const char*s2)
{
    for(;*s1==*s2;s1++,s2++)
        if(*s1=='\0') return 0;
    return (*s1-*s2)
}
```

指针数组

数组元素的类型可以是基本类型,也可以是指针类型,**一个以指针类型数据为元素的数组称为指针数组**,数组中的每个元素用于保存地址.

指针数组的概念

指针数组的声明

指针数组的声明形式为:

```
type *array_name[size];
```

其中:

- `type` 是指针所指向的数据类型;
- `*` ,是指针说明符,表示**数组元素的类型是指针**;
- `array_name` 是数组的名称;
- `size` 是数组的大小,即数组中指针的个数.

例如,一个**包含5个指向整数的指针的数组**可以声明为: `int *ptrArray[5];` ,其中[]的优先级高于*,所以先解释 `ptrArray[5]` ,说明ptrArray是一个有5个元素的数组;然后解释 `int *` ,说明数组的每个元素是一个int类型指针.简称ptrArray是有5个元素的整型指针数组.
注意 `int *ptrArray[5];` 不同于 `int (*ptrArray)[5];` ,后者定义了一个**指向包含5个整数的数组的指针**.

指针数组是一种非常有用的数据结构,其作用包括以下几个方面:

1. **多个指针的管理**:指针数组允许同时管理多个指针.这对于需要处理多个数据元素或对象的情况非常有用.可以使用指针数组来管理多个字符串,或者管理多个动态分配的内存块.
2. **动态内存分配**:指针数组可以用于动态分配内存,例如创建一个指针数组,每个指针指向动态分配的内存块.这对于处理动态大小的数据结构或者在运行时需要分配不确定数量内存的情况非常有用.
3. **函数参数传递**:指针数组可以作为函数参数进行传递,这样可以在函数内部修改指针数组所指向的数据.通过传递指针数组,可以有效地传递和处理多个数据对象,而不需要将它们全部复制一遍.

4. **多维数组**:指针数组还可以用于创建多维数组.通过将指针数组的每个元素设置为另一个指针数组,可以创建多维数据结构,例如二维数组或更高维度的数组.其优点在于二维数组的每一行或字符数组的每一个字符串可以有不同的长度.
5. **数据结构的实现**:指针数组可以用于实现各种数据结构,例如树,图,链表等.通过将指针数组的元素设置为指向其他数据结构的指针,可以构建复杂的数据结构,实现各种算法和数据操作.

指针数组的初始化

E.G.

```
int a[] = {1,2,3,4};
int b[] = {5,6,7};
int *p[2] = {a,b};
```

说明p是一个有两个元素的int型指针数组,p[0]初始化为a,p[1]初始化为b.即p[0]指向数组a的首元素,p[1]指向数组b的首元素.因此指针数组p描述了一个二维数组.

`p[1][2]`，`*(p[1]+2)`，`*(*(p+1)+2)` 都表示b[3]=7.参考链接.p[1]的值是b,b[3]的值和*(b+2)一样.在指针数组中,p指向p[0]=a,(p+1)指向p[1]=b,则*(p+1)的值就是b,(b+2)指向的是b[3].

事实上,p,(p+1)可以认为是指针的指针,地址的地址.*p,*(p+1),p[0],p[1]得到的结果还是指针(地址).

用指针数组表示二维数组时,需要额外增加用作指针的存储开销,但指针数组较为灵活.

考虑三角矩阵:

a_{00}

a_{10} a_{11}

...

a_{40} a_{41} a_{42} a_{43} a_{44}

定义一个有15个元素的**一维数组来存储三角矩阵元素**,定义一个有5个元素的**指针数组存储每一行的首地址**:

```
int a[15],*p[5];
for(i=0;i<5;i++)
    p[i] = a+i*(i+1)/2;
/*
for(sum=0,i=0;i<5;i++){
    p[i] = a+sum;
    sum+=i+1;
}
 */
```

指针数组不常用于描述整型,浮点型等二维数组,应用最多的是描述字符串数组,尤其是**不同长度的字符串组成的数组**

用指针变量表示字符串数组

字符指针数组的初始化

双引号 "" 表示一个字符串字面量,在C语言中用于表示一个字符数组,这些字符串字面量本身就是指向字符的指针,可作为字符串常量使用.
E.G.

```
int isKeyword(char *s)
{
    static char *keyword[] = {"auto", "char", "continue", "if", "int", NULL};
    /* 关键字表 */
    int i;
    for (i = 0; keyword[i] != NULL; i++)
        if (!strcmp(s, keyword[i]))
            return 1;
    return 0;
}
```

keyword是一个有6个指针类型元素的字符指针数组,初值表由关键字**字符串的首地址**组成.例如元素keyword[0]是指向第0个串"auto"的指针.keyword数组只存放字符串首地址,而字符串存放在**常量区**.
keyword[0]可以视为第0个字符串, `*keyword[0]` 和 `keyword[0][0]` 等效, `*(keyword[0]+1)` 和 `keyword[0][1]` 等效.

```
int searchKeywords(const char *keywords[], int numKeywords, const char *str, int foundIndexes[]
    int count = 0; // 找到的关键字数量
    for (int i = 0; i < numKeywords; i++) {
        if (strstr(str, keywords[i]) != NULL) {
            foundIndexes[count++] = i; // 将找到的关键字的索引存储在数组中
        }
    }
    return count; // 返回找到的关键字数量
}
```

指针数组可以作为函数形参,这个函数中还设置了数组 `int foundIndexes[]` 作为返回值.

指针数组与二维数组的比较

字符串数组也可以用二位数组表
示: `char keyword2[][19] = {"auto", "char", "continue", "if", "int", NULL};` .
指针数组和二维数组的差别在于:

1. 二维数组建立了所有行长度都相同的数组(有冗余);指针数组建立的是一个不规则数组,每个字符串占据的占据的内存空间只是它本身的长度,但是指针数组本身要占用空间.
2. keyword和keyword2的类型不同,keyword是一个**char指针数组**,里面存放6个地址值,其中元素是类型为char *的变量;而keyword2是一个**char数组的数组**,存放有6个完整的字符串.
3. 元素keyword[i]的类型是**char *的变量**;keyword2[i]表示第i个字符串的首地址,其类型是**char *的常量**. 用二维数组代替指针数组时,循环条件要改成 `*keyword[i]!='\0'` .

动态分配字符串数组

如果字符串事先不确定,需要由用户输入,这时无法通过初始化的形式创建字符串数组.下列代码有严重问题:
E.G.

```
char *str[10];
int i;
for(i=0;i<10;i++) scanf("%s",str[i]);
```

str数组中的元素没有赋值,其值随机地指向某个内存单元,通过scanf函数赋值,实际上是给str[i]随机指向的存储单元赋值.

- **指针数组的每个元素只能存地址值,用来保存各个字符串的首地址,不能存字符串本身**.指针数组的声明仅仅获得了**存放地址的空间**,并没有分配存放字符串的空间.
- 利用**动态存储技术**可以实现字符串数组的无冗余存储,根据实际输入的字符串长度用malloc函数分配存储空间.
- 动态存储分配函数是C语言的标准函数,用来**管理堆内存**,相关函数有malloc,calloc,realloc,free等,这些函数的原型声明在头文件<stdlib.h>中.
    - `void *malloc(size_t size);` 分配size字节
    - `void *calloc(size_t n,size_t size);` 分配n*size字节
    - `void *realloc(void *p,size_t size);` 把p所指的已分配存储区改为size字节
    - `void free(void *p);` 释放p所指向的存储区
      其中的size_t是unsigned int的别名,表示无符号整型,在<stdio.h>通过关键字typedef定义.

1. malloc函数用于**向系统申请分配size字节的在堆(heap)中的连续存储空间**.如果分配成功,则**返回分配区域的首地址**;如果分配失败(如内存容量不足),则返回NULL.分配的内存空间是未初始化的,其内容是

未定义的.因此,在使用之前,应该对分配的内存进行初始化.

2. 当分配的内存不再使用时,应该用free(p)函数释放由p所指的内存,并将它返回给系统,以便这些内存成为再分配时的可用内存,防止内存泄漏.**p是由malloc,calloc,realloc分配的存储区的指针**,一旦释放,就不能用p引用存储区中的数据.
   以下是几个内存泄漏的可能情况:
   i. **循环分配内存而不释放:** 如果在程序中重复执行 `malloc()`,但没有对应的 `free()` 调用来释放内存,每次分配的内存都会累积,最终导致内存耗尽.
   ii. **在函数中动态分配内存但没有释放:** 如果在函数内部使用 `malloc()` 分配内存,但在函数返回之前没有释放该内存,则该内存将会泄漏.
   iii. **指针赋值导致无法访问的内存:** 如果一个指针被赋值为 `malloc()` 分配的内存地址,但之后该指针被重新赋值或丢失,导致无法访问该内存并释放它,这也会造成内存泄漏.
3. malloc的返回值类型是void *,void *表示未确定类型的指针,是指malloc在分配存储空间时还不知道用户用这段空间来存储什么类型的数据.不能对void指针施加单目*运算,但void *类型可以**通过类型转换符强制转换为其他确定类型的指针**.

```
/* 用malloc函数动态创建一个有n个元素的int数组 */
int i,n,*p;
scanf("%d",&n);
if((p=(int *)malloc(n*sizeof(int)))==NULL){
    printf("Out of memory!")
    exit(-1)//程序以异常状态退出,并且返回值为 -1
}
for(i=0;i<n;i++) scanf("%d",p+i);
// 在程序结束时释放动态分配的内存
free(p);
return 0;
```

在引用p指向的动态存储区中的数据之前,应该先判断p是否为空.如果p为空,则说明动态存储分配失败,应该输出提示并结束程序.

1. 上面我们利用malloc函数动态创建了有n个元素的数组,而事实上,C99允许直接动态创建数组,即**可变长度数组(VLA)**,它允许在函数内部声明数组时使用变量作为其大小.VLA 的大小**在运行时确定,而不是编译时确定**(静态数组的大小在编译时就确定了),这使得可以根据程序的需要动态分配数组的大小.可变长度数组只能在函数内部声明,而不能作为全局变量;另外,它们不是动态分配内存的方式,因此不会触发内存分配失败的错误处理机制,而是依赖于**栈空间的可用性**.

```
int i,n,*p;
scanf("%d",&n);
int a[n];
for(i=0;i<n;i++) scanf("%d",p+i);
```

依据程序运行时所输入的n值,创建有n个元素的数组,后面n的值发生变化,a的大小不会发生变化.

5. malloc函数可以用于动态分配多维数组.这在处理复杂的数据结构时非常有用.

```
int rows, cols;
printf("Enter the number of rows and columns:\n");
scanf("%d %d", &rows, &cols);
int **matrix = (int **)malloc(rows * sizeof(int *));//指向指针的指针
for (int i = 0; i < rows; i++) {
    matrix[i] = (int *)malloc(cols * sizeof(int));
}
// 使用二维数组 matrix
for (int i = 0; i < rows; i++) {
    free(matrix[i]); // 释放每一行的内存
}
free(matrix); // 释放存储指针的内存
```

6. 用动态存储技术可以实现字符串数组的无冗余存储,要用到一些字符串处理函数

```
int strLength, i;
char *s[STRNUM],temp[MAX_LEN];
printf("输入%d个字符串:\n", STRNUM);
for (i = 0; i < STRNUM; i++) {
    fgets(temp, MAX_LEN, stdin);
    // 每次循环迭代中,fgets() 函数会读取新的字符串并将其存储到 temp 数组中,覆盖掉之前的内容.
    strLength = strlen(temp);
    if (temp[strLength - 1] == '\n') // 如果输入的字符串末尾有换行符
    temp[strLength - 1] = '\0'; // 将其替换为字符串结束符

    s[i] = (char *)malloc(strLength + 1);//动态无冗余存储
    if (s[i] == NULL) {
        // 释放已分配的内存
        for (int j = 0; j < i; j++) {
            free(s[j]);
        }
        printf("Memory allocation failed!\n");
        return -1;
    }
    strcpy(s[i], temp);
    //释放malloc所分配的空间,参数是指针
    for (i = 0; i < STRNUM; i++) {
        free(s[i]);
    }
}
```

指针数组作函数参数

```
void strSort(char *str[],int size)
{
    char *temp;
    int i,j;
    for(i=0;i<size-1;i++)
        for(j=0;j<size-i-1;j++)
            if (strcmp(str[j],str[j+1])>0)
            {
                temp=str[j];
                str[j]=str[j+1];
                str[j+1]=temp;
            }
}
```

strSort函数的形参声明为char *str[],调用时实参是字符指针数组名s,数组名s是首元素s[0]的地址,等价于&s[0].s本身是指针,它所指向的对象s[0]也是char *指针类型,这种**指向指针的指针称为二级指针**.因char *str[]等价于char **str.

指向指针的指针

二级指针的声明

指针变量也有地址,**存放指针变量地址的变量称为指向指针的指针,或一级指针的指针,或二级指针**.二级指针在说明时变量名前面有两个*.

`char **p;` ,其中右起第一个*说明p是指针变量,第二个*说明p指向的对象是指针(一级指针),char说明一级指针指向的对象是字符型数据.

例如 `int x=1234,*p=&x,**pp=&p;` ,p是int指针,里面存放了变量x的地址,*p代表x.pp是指向int指针的指针(二级指针),里面存放指针变量p的地址.*pp代表p,**pp就是*p代表x.因此,利用二级指针间接访问其最终指向的对象需要施加两次单目*运算.

二级指针与制作数组

对于指针数组,数组元素的类型是指针,需要声明一个指向指针的指针来引用数组元素.例如:\

```
char *keywords[] = {"auto", "char", "continue", "if", "int",NULL};
char **p=keyword;/* keyword等价于&keyword[0] */
```

其中,p是二级指针,指向指针数组首元素keyword[0],而keyword[0]本身又是字符指针,所以p应该声明为二级指针. `*(p+i)` 等价于keyword[i], `*(*(p+i)+j)` 等价于keyword[i][j].

```
int i;
for (i = 0; keyword[i] != NULL; i++)
    if (!strcmp(s, keyword[i])) return 1;
return 0;
/* 可以改写为: */
char **p;
for (p=keyword; *p!= NULL; p++)
    if (!strcmp(s, *p)) return 1;
return 0;
```

二级指针p初始指向元素keyword[0],*p代表keyword[0].指向p++后,p指向下一元素keyword[1],*p就代表keyword[1].

二级指针参数

参见文件**D:\C_code\C\Pointer\pointer_8\pointer_8(字符串的字典序排序).c**其中重点在于指针数组的动态内存分配. `s=(char **)malloc(sizeof(char *)*STRNUM);`

main函数的参数

命令行参数

命令行界面是图形界面普及之前的最广泛界面,要用键盘输入指令.命令行界面要求用户记忆操作的命令,更节约计算机资源.
命令行是在命令行模式(windows的cmd)下,用户输入的用于运行程序的行,在命令行中,空格隔开的字符串就是命令行参数.假定有一个名为copy的程序,在Windows下运行该程序的命令行为:
c:\>copy abc.txt def.txt
该命令行有三个参数,参数copy是**可执行程序名**,其后的abc.txt和def.txt是程序执行需要的参数.该命令行执行copy程序,将copy,adc.txt和def.txt三个参数传递给程序的main函数,实现将文件abc.txt的内容赋值到文件def.txt.因此该main函数中带有形参.

带参main函数的定义

main函数具有两个形参,第一个称为argc(argument count),类型为int,表示**命令行参数的个数**;第二个称为argv(argument value),是一个字符指针数组,**数组的元素分别指向命令行中用空格分隔的参数**.其定义形式为:

```
int main(int argc,char *argv[])
{
    /* 函数体 */
}
```

对于命令行 `c:\>copy abc.txt def.txt` ,相当于将3传给argc,指针数组argv中有三个元素,argv[0]指向串"abc.txt",argv[1]指向串"def.txt".

```
#include<stdio.h>
int main(int argc,char **argv)
{
    while(--argc>0)
        printf("%s%c",*++argv,(argc>1)?' ':'\n');
    return 0;
}
```

编译链接后可执行文件的名称为 `D:\C_code\C\Pointer\pointer_11\pointer_11_commandline.exe` ,在命令行环境下输入下面命令行: `D:\C_code\C\Pointer\pointer_11\pointer_11_commandline.exe I like you` ,则输出为 `I like you` .命令行参数的个数是4,argc的值是4, `char **argv` 等价于 `char *argv[]` ,*argv相当于argv[0],*(argv+1)相当于argv[1].

命令行参数的传递

执行上述程序时,有4个参数,先将4传递给形参argc,命令行中每个参数作为字符串被存储到内存中,并且分配一个**指针数组集**存放4个字符串的首地址,指针数组的末尾是一个NULL指针,因此指针数组的大小是5.然后将指针数组的首地址传递给形参argv,使argv指向指针数组第零个元素.

指针函数

C语言的函数可以返回除数组之外的任何类型数据.如果函数的返回值是一个指针,该函数称为**指针函数**.

指针函数的声明

函数指针声明的一般形式是:

```
return_type *function_name(parameter_list);
```

其中:

- `return_type` :指针函数的返回类型,即指针所指向的数据类型.
- `function_name` :指针函数的名称,括号表明这是一个函数.
- `parameter_list` :指针函数的参数列表,包括参数的类型和名称.如果函数不接受任何参数,则可以留空或使用 `void` 表示.
- *表明此函数的返回值是一个指针.事实上,*的优先级低于()的优先级,意味着函数名先代表一个函数,其次其返回值是个指针.

指针函数的声明也可以包含一些修饰符,如 `static` , `extern` , `const` 等,用于指定函数的存储类别或其他属性.修饰符通常放置在返回类型之前.例如: `extern int* getPointer(int x);` .在这个声明中, `extern` 表示该函数在其他文件中定义,而不是在当前文件中定义.

```
char *myStrcpy(char *t, const char *s)
{
    while (*t++ = *s++);
    return t;
}
```

由于函数返回值是字符型指针,可以有语句: `printf("str1:%s\n",myStrcpy(str1,str2));`

指针函数返回值分析

用指针作为函数返回值时要注意,函数运行结束后会销毁函数内部定义的所有局部数据,包括自动变量,自动数值,和形式参数.返回值的指针不要指向这些数据.

对比下列几个程序

```
char *getString(void)
{
    char *p="Hello world!";
    return p;
}
```

`char *p="Hello world!";` 定义的字符串常量存储在**静态存储区(常量区)**,而局部指针变量存储在栈上.在函数返回时,栈上的局部变量生命周期结束,内存空间被释放.因此,在函数返回后,**指针变量 p 所占据的栈上的内存会被销毁,但它指向的字符串常量在静态存储区中的内存不会受到影响**.虽然局部变量指针p占据的内存被释放,但是函数仍然返回了一个指针,这个指针指向常量区字符串首地址,可以被视为指向常量的指针(也是指针常量),用 `printf("%s",getString());` 调用时仍然起作用.

```
char *getString(void)
{
    char p[]="Hello world!";
    return p;
}
```

这个程序中,**字符串(数组)也存储在栈上,都是局部变量**,(栈是一个后进先出(LIFO)的数据结构,函数返回时会弹出栈帧,从而释放局部变量的内存空间)函数运行完后,存储他们的空间全部释放.虽然函数仍然返回了一个地址,但这个地址已经不能访问了,系统提醒:"function returns address of local variable".

```
char *getString(void)
{
    static char p[]="Hello world!";
    return p;
}
```

将p定义为**局部静态数组,存储在静态区**.函数返回后,字符数组的空间不会被释放,返回的地址仍然有效.调用函数时,仍然可以访问函数返回的指针所指向的数组.

```
char *getString(void) {
    char *p = malloc(strlen("Hello world!") + 1);
    if (p != NULL) {
        strcpy(p, "Hello world!");
    }
    return p;
}
```

该程序需要使用一些头文件,运用动态内存分配技术,我们在堆区为指针p分配了无冗余的空间,而**堆上分配的内存不会在函数结束时自动释放**,只能手动释放.因此,仍然可以用返回的指针值访问字符串.

指针函数的定义与应用

```
char *myStrstr(const char *str,const char *substr)
{
    const char *pstr, *psub, *res;
    for(;*str!='\0';str++){
        res=str;
        for(pstr=str,psub=substr;*pstr!='\0';pstr++,psub++)
            if(*pstr!=*psub) break;
        if(*psub=='\0') return res;
    }
    return NULL;
}


char *strSource="...";
    char *strFind="...";
    char *p=strSource;
    int count=0,len;
    len=strlen(strFind);
    while((p=myStrstr(p,strFind))!=NULL){
        printf("%s\n",p);
        p+=len;
        count++;
    }
```

while循环语句涵盖了指针函数和指针作为参数的函数的本质特点,其**调用和返回的都是指针(地址),而非数组**.

指向函数的指针

函数指针变量的声明

函数入口的地址可以存储在某个函数指针变量中,可以通过这个函数指针变量调用所指向的函数.函数指针变量的声明形式为:

```
类型区分符 (*标识符)(形参表);
```

- (*标识符) 说明该标识符是一个指针变量名,其中*是指针说明符;
- (形参表) 是函数的标记,表示这个指针指向函数;
- 形参表是所指函数的形式参数声明.类型区分符表示所指函数的返回类型.

例如, `int (*comp)(char*,char*);` 表示:**comp是指向有两个char*参数的int函数的指针**,不能写成 `int *comp(char*,char*);` ,后者是一个指针函数.

函数返回值类型和形参列表共同决定了函数指针的类型,不同类型的函数指针不能混用.假如函数原型为: `int func(int,int);` ,则用来指向函数func的指针应该声明为: `int (*pf)(int,int)=func;` ,函数指针的初始化也可以用赋值语句来进行: `pf=func` .

在函数指针被声明并初始化之后,可以用函数指针调用函数:

`(*pf)(a,b);` 或 `pf(a,b);`

- 因为**pf指向func函数**,(*pf)就代表访问func函数,因此 `(*pf)(a,b)` 和 `func(a,b)` 等价;
- 又因为**函数名就是函数入口的地址**,因此pf等价于func,从而 `pf(a,b)` 和 `func(a,b)` 等价.

函数指针的应用

1. 作为参数传递给其他函数;
2. 用于散转程序.
   一个函数通过运行时决定的指针来调用另一个函数的行为称为**回掉**.

指针与多维数组

指向数组元素的指针

```
int a[2][3]={{1,3,5},{2,4,6}},*p;
for(p=&a[0][0],i=0;i<2;i++)
{
    for(j=0;j<3;j++)
        printf("%3d",*(p+3*i+j));
    printf("\n");
}
```

`p+3*i+j` 指向a[i][j], `*(p+3*i+j)` 表示a[i][j].
表达式 `*(p+3*i+j)` 也可以改为 `*p++` ,后者会改变p的值.

指向数组的指针

二维数组相当于一个特殊的一维数组,里面每个数组又是一维数组.
以 `int a[2][3];` 为例,a可以看成由两个行元素a[0]和a[1]组成的一维数组,a[0]和a[1]又都是包含三个整数的一维数组.也就是说**a[0]和a[1]是两个一维数组的数组名**,表示一维数组的首地址,即a[0]是第零行首元素的地址,等价于&a[0][0],是一个指针;a[1]是第一行首元素的地址,等价于&a[1][0].

二维数组名a也是数组的首地址,虽然**与一维数组名a[0]的值一样,但是指针类型与之不同**,二维数组名a不是指向int元素的指针,所指的对象不是a[0][0].二维数组名是**指向行元素的指针**,所指的对象是整个0行,a+1指向下一行,即第1行.

由于a指向第0行,***a就是a[0],均是第0行的首地址**,指向a[0][0];a+1指向第1行,因此*(a+1)相当于a[1],均是第一行首地址,指向元素a[1][0].

由于*a和a[0]等价,因此**a和*a[0]同a[0][0];*a+1和a[0]+1均指向a[0][1],因此*(*a+1)和*(a[0]+1)同a[0][1].注意,*a+1不同于*(a+1).于是,可以通过数组名a和行首指针a[1]来引用二维数组元素.

```c
int main()
{
    int a[2][3]={};
    printf("Address of a[0][0]: %p\n", (void *)&a[0][0]);        // a[0][0]的地址
    printf("Value of a:         %p\n", (void *)a);               // a指向第0行
    printf("Value of a[0]:      %p\n", (void *)a[0]);            // a[0]指向第0行第0个元素
    printf("Value of *a:        %p\n", (void *)*a);              // *a等价于a[0],指向第0行第0个元
    printf("Value of &a:        %p\n", (void ***)&a);            // a的地址
    printf("Value of &a[0]:     %p\n", (void **)&a[0]);          // a[0]的地址
}
```

用typedef定义类型名

复杂声明