《计算机语言与程序设计》
第8周 指针第二讲

清华大学 自动化系
范 静 涛

数组名，大不同！！！

int a[4];

表达式	正确?	含义	类型
a	正确	数组名，与&或sizeof联用，表示整个数组	int[4]
a	正确	数组名，其他情况下，表示首元素指针&a[0]，整型指针常数	int* const
&a	正确	数组名与&连用表示整个数组，表达式为整个数组的指针常量	(int[4])* const
a + i	正确	a表示&a[0]。表达式含义是&a[i]，第i个元素的指针，整型指针常数	int* const
&a[i]	正确	第i个元素的指针，整型指针常数	int* const
a[i]	正确	第i个元素，整型	int
*(a + i)	正确	对(第i个元素的指针)去引用，第i个元素，整型	int

数组名，大不同！！！

int b[3][4];

表达式	正确?	含义	类型
b	正确	二维数组名，与sizeof联用，表示整个数组	int[3][4]
b	正确	二维数组名，其他情况下，表示当做一维数组时的首元素指针&b[0]，指向(4元素int数组)的指针常数	(int[4])* const
&b	正确	数组名与&连用表示整个数组，表达式为整个数组的指针常量	(int[3][4])* const
b[i]	正确	一维数组名，与sizeof联用，表示整个数组	int[4]
b[i]	正确	一维数组名，其他情况下，表示首元素指针&b[i][0]，整型指针常量	int* const
b + i	正确	&b[i]，第i行的指针	(int[4])* const
&b[i]	正确	第i行的指针	(int[4])* const
b[i] + j	正确	&b[i][j]，第i行第j列元素的指针常量。	int* const
*(b+i)+j	语法正确 逻辑需要转义	*(b+i)表示第i行，是一维数组的名字，转义为其首元素指针，类型是int* const，+j类型不变，是第i行第j列元素的指针常量	int* const
&b[i][j]	正确	第i行第j列元素的指针常量	int* const
*(b[i]+j)	正确	b[i]代表首元素指针&b[i][0]。整体表示b[i][j]	int
((b+i)+j)	语法正确 逻辑需要转义	*(b+i)表示第i行，是一维数组的名字，转义为其首元素指针，类型是int* const，+j类型不变，是第i行第j列元素的指针常量，去引用为整型	int

指针访问二维数组：指向数组元素

假设，ROWS和COLS是大于0的整型常数

```
int a[ROWS][COLS]
int* p = &a[0][0];
```

访问一个二维数组元素a[row][col]，可以用：

- 数组下标法：a[row][col]
- 指针下标法：p[row * COLS + col]
- 指针引用法：*(p + row * COLS + col)

由于指针变量p指向类型为int，说明p指向的一定是整型实体(a的元素)。当通过p
来访问二维数组或多维数组时，本质上是将多维数组按一维数组来处理，

指针访问二维数组：指向一维数组

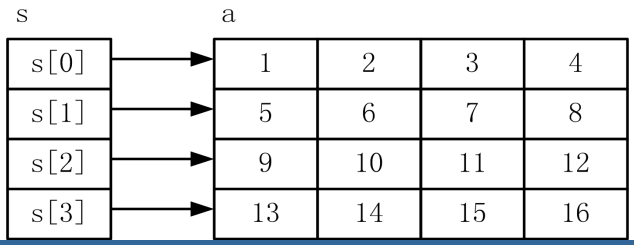
指向数组的指针数组，可以方便地处理高维数组

- **指向数组的指针**：
  int (*p)[4];或者 typedef int arr[4]; arr* p;
  p指向有4个元素的数组，每个元素为整型

- **元素是指针的数组**：int* p[4]
  表示p有4个元素，每个元素是 int*（指向整型变量的指针变量）

初始化：已知内存或者空指针

```
//二维数组
int a[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
//一维指针数组初始化
int* s[4] = {&a[0][0], &a[1][0], &a[2][0], &a[3][0]};
```



s[0]：指向a[0][0]

*s[0]：a[0][0]

s[i] + j：指向 a[i][j]

(s[i] + j)、s[i][j] 、(*(s+i)+j)：a[i][j]

指针访问高维数组

a[1][1][1]的含义?

a[1][1]的含义?

a[1]的含义?

a的含义?

a + 1的含义?

*a的含义?

*(a + 1)含义?

*a + 1含义?

**a的含义?

**(a + 1)含义?

*(*a + 1)含义?

((*a + 1))含义?

指针访问高维数组

接上页的思考，如果需要定义指针

与a同级的指针p1如何定义？ p1++ 含义

与*a同级的指针p2如何定义？ p2++ 含义

与**a同级的指针p3如何定义？ p3++ 含义

p1、p2、p3的换算关系是什么？

二级指针与一级指针数组

例如：

```
int a[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
//int* s[4] = {a[0], a[1], a[2], a[3]};
int* s[4] = {&a[0][0], &a[1][0], &a[2][0], &a[3][0]};
//int** pp = s;
//int** means: (int*)* or pointer of (pointer of int)
int** pp = &s[0];
```

- 二级指针pp指向一维指针数组s的首元素地址

pp = s 或 pp = &s[0]

- 二级指针pp指向一维指针数组s[i]

pp = s + i、pp = &s[i] 或 pp + i

- *(pp + i)、pp[i]等价于s[i]

二级指针与一级指针数组

例如：

```
#include <stdio.h>
int main(void) {
    char* name[] = {
        "follow me",
        "basic",
        "great wall",
        "fortran",
        "computer design"
    };
    char** p;
    for (int i = 0; i < 5; i++) {
        p = name + i; //二级指针行地址
        printf("%s\n", *p); //*p相当于一行数据
    }
    return 0;
}
```

二级指针与二维数组

例如

```
int a[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
//int* s[4] = {a[0], a[1], a[2], a[3]};
int* s[4] = {&a[0][0], &a[1][0], &a[2][0], &a[3][0]};
//int** pp = s;
//int** means: (int*)* or pointer of (pointer of int)
int** pp = &s[0];
```

通过二级指针访问一维指针数组s再间接访问二维数组a的典型形式为：

*(pp + i)、pp[i]：指向a[i][0]

**(pp + i)、*pp[i]、*(pp[i] + 0)、pp[i][0]：a[i][0]

*(pp + i) + j、pp[i] + j：指向a[i][j]

((pp + i) + j)、*(pp[i] + j)、pp[i][j]：a[i][j]

字符数组与字符指针

C语言可以定义一个字符数组，用字符串常量初始化它

```
char str[] = "C Language"; //str被分配了数组空间
```

也允许定义一个字符指针，初始化时指向一个字符串常量，或者用字符串常量赋值给字符指针变量p。为字符串常量分配了存储空间，以数组形式存储，并在字符串末尾追加一个结束符'\0'。首字符地址存储在p中:

```
char* p = "C Language"; //p中仅存储字符串常量首地址
```

或者

```
char* p;
p = "C Language"; //p中仅存储字符串常量首地址
```

字符数组与字符指针

char str[] = "C Language"; **//str被分配了数组空间**

char* p = "C Language"; **//p中仅存储字符串常量首地址**

表达式	是否正确?
str++	错误，str是指针常量
p++	正确，p是指针变量
*(str + i) = 'a'	正确，str的元素是字符变量
*(p + i) = 'a'	错误，p指向的是字符常量
str[i] = 'a'	正确，str的元素是字符变量
p[i] = 'a'	错误，p指向的是字符常量
str = "123"	错误，不能给数组整体赋值
p = "123"	正确，用字符串常量首地址给指针变量赋值

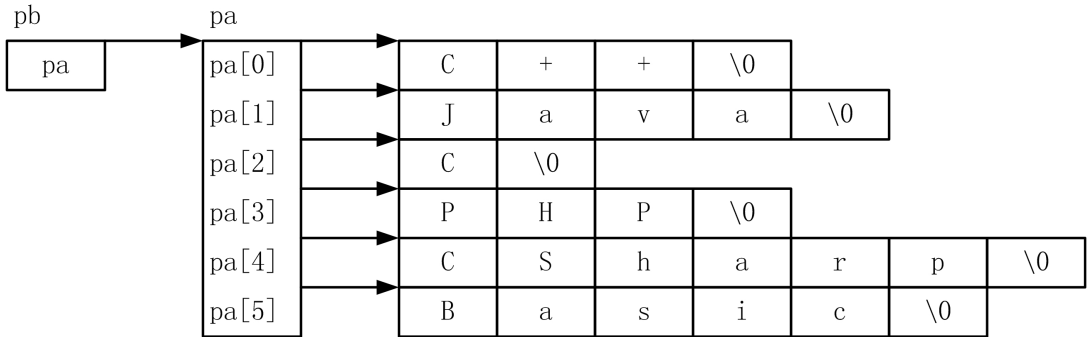
字符串数组与字符串指针

由于一个字符指针可以指向一个字符串，为了用指针表示字符串数组，有两种方案

- 使用**指针数组**，例如 `char *pa[6] = {"C++", "Java", "C", "PHP", "Csharp", "Basic"};`

  其中pa为一维数组，有6个元素，每个元素均是一个字符指针。

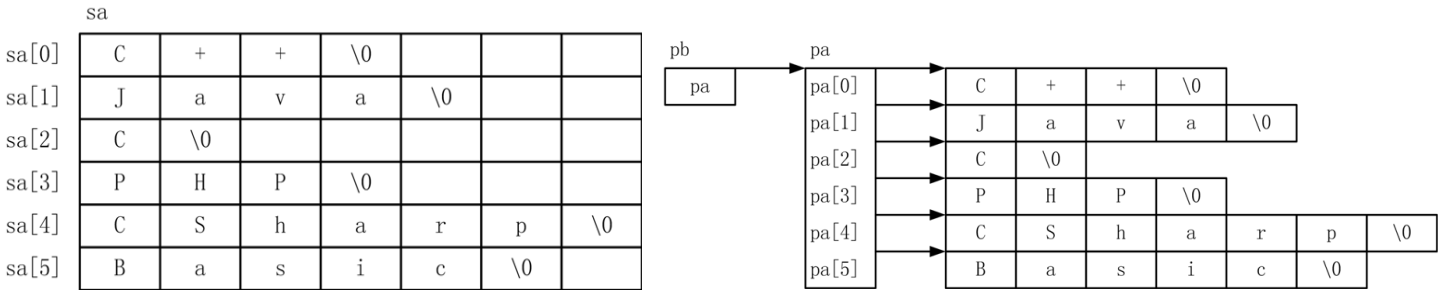
- 定义**指向指针变量的指针变量**。 `char **pb = &pa[0];`



字符串数组与字符串指针

用字符串数组存储若干个字符串时，要求每行包含的元素个数相等，因此需取最大字符串长度作为列数，往往浪费内存单元

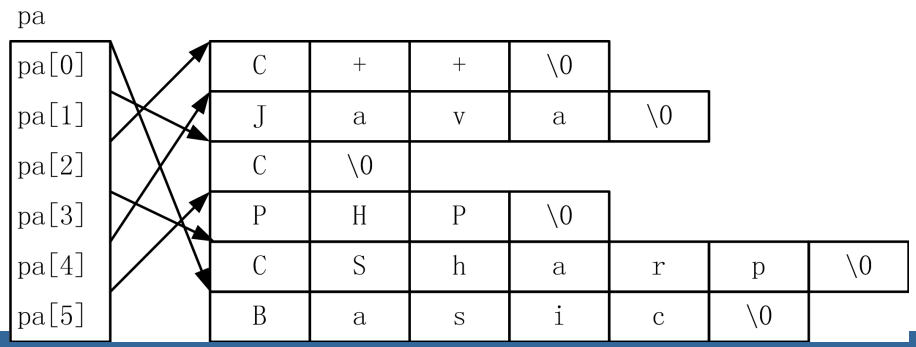
若使用字符指针数组，各个字符串按实际长度存储，指针数组元素只是各个字符串的首地址，不存在浪费内存问题



字符串数组与字符串指针

利用冒泡法对字符串排序

```
#include <stdio.h>
#include <string.h>
int main(void) {
    char* PointerArr[6] = {"C++", "Java", "C", "PHP", "CSharp", "Basic"};
    char* TempPointer;
    for (int j = 0; j < 6 - 1; j++) {
        for (int i = 0; i < 6 - 1 - j; i++) {
            if (strcmp(PointerArr[i], PointerArr[i + 1]) > 0){
                //指针交换
                TempPointer = PointerArr[i];
                PointerArr[i] = PointerArr[i + 1];
                PointerArr[i + 1] = TempPointer;
            }//end if
        }//end i loop
    }//end j loop
    return 0;
}
```



字符串数组与字符串指针

字符串指针数组的一个重要应用是作为main函数的形参。

main函数可以有参数，形式如下：

```
int main(int argc, char* argv[])
```

命令行的一般形式为：

命令名　　参数1　　参数2

执行上述命令行时，系统会将命令行各个参数传递到main函数：

- argc：命令行中字符串的个数（含可执行程序名称）；
- argv[0]：可执行程序名称字符串的首地址；
- argv[1]：参数1字符串的首地址；
- argv[2]：参数2字符串的首地址，其余以此类推。

字符串数组与字符串指针

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf("program:%s\n", argv[0]); //输出程序名
    for (int i = 1; i < argc; i++) {
        printf("%s\n", argv[i]); //输出程序参数字符串
    }
    return 0;
}
```

假定程序取名TEST，在命令行提示符中输入以下命令

C:\>TEST IN.DAT OUT.DAT

程序运行结果如下：

program:TEST

IN.DAT

OUT.DAT

指针作为函数参数

类似数组作为函数参数，函数调用时发生**地址的值传递**

指针参数能够**将更多的运算结果返回到主调函数中**。指针是函数间参数传递的重要工具。

一般形式

```
返回类型 函数名(指向类型* 指针变量名,……)
{
    函数体
}
```

注意：C语言不会对任何指针类型做隐式类型转换，因此**函数形参与实参的指针指向类型必须严格一致**。

指针作为函数参数：保留被调函数的结果

输入两个整数，按从小到大次序输出，用函数实现。

```
#include <stdio.h>
void swap(int* p1, int* p2) {
    int t;
    t   = *p1;
    *p1 = *p2;
    *p2 = t;
}


int main(void) {
    int a;
    int b;
    scanf("%d%d",&a,&b);
    if (a > b) {
        swap(&a, &b);
    }
    printf("min=%d,max=%d\n",a,b); //输出
    return 0;
}
```

如果需要在函数调用结束后保留被调函数的计算结果，需要

用指针变量作为形参，

将变量的指针（或地址）传递到被调函数中

通过指针dereferencing
达到修改变量的目的。

指针作为函数参数：返回多个结果

编写函数，计算并返回a和b
的平方和、自然对数和、几何平均数、和的平方根。

```
#include <stdio.h>
#include <math.h>
double fun(double a, double b, double* sqab, double* lnab, double* avg) {
    *sqab = a * a + b * b; //*sqab返回平方和
    *lnab = log(a) + log(b); //*lnab返回自然对数和
    *avg = (a + b) / 2; //*avg返回几何平均数
    return (sqrt(a + b)); //函数返回和的平方根
} //↑绝对绝对绝对，不要这么写，因为函数功能要保持单一！！！
int main(void) {
    double x=10,y=12,fsq,fln,favg,fsqr;
    fsqr = fun(x, y, &fsq, &fln, &favg);
    printf("%lf,%lf,%lf,%lf,%lf,%lf\n", x, y, fsq, fln, favg, fsqr);
    return 0;
}
```

指针作为函数参数：与数组作函数参数的异同

作用和写法类似

- 函数调用中，实参向形参均传递数据地址，主调函数与被调函数指向同一段内存单元
- 数组与指针对于数据元素的引用写法一致：下标法、间接引用等,一般以参数为准

均无法做有效数据长度判断

```
//编写函数average，返回数组n个元素的平均值。
#include <stdio.h>
//等价于average(double a[], int n)
double average(double* a,  int n) {
    double avg = 0.0;
    for (int i = 0; i < n; i++) {
        avg += *(a + i);
    }
    return avg / n ;
}
int main(void) {
    double x[10] = {12, 6.5, 9,81, 1.5, 81, 9, 90.5, 78, 88};
    printf("average=%lf\n", average(x,10));
    return 0;
}
```

指针作为函数参数：与数组作函数参数的异同

含义不同

- 数组名是一个指针常量，而指针名是一个指针变量。但作为参数是并不会有区分。

```
void fun(double* A, double B[100]) {
    A++; //正确，A是指针变量
    B++; //错误，B是指针常量，不能做自增自减
    A = B; //正确，A是指针变量，可以重新指向B
    B = A; //错误，B是指针常量，不能被赋值
}
```

编译器认为以上都正确

指针作为函数参数：实例

用选择法对10个整数按由大到小顺序排序。

```
#include <stdio.h>
const int LEN = 10;
int main(void) {
    int a[LEN];
    int* p = a;
    // 读入数组
    for(int i = 0; i < LEN; i++) {
        scanf("%d",p);
        p++;
    }
    // 排序，这里p已迷路，重新指向a
    p = a;
    sort(p, LEN);
    //打印，p再次重新指向a
    for(p = a, i = 0; i < LEN ;i++) {
        printf("%d",*p);
        p++;
    }
}
```

```
void  sort(int x[ ], int n) {
    int k;
    int t;
    for(int i = 0; i < n - 1; i++)
    {
        k = i;
        // 选择i个元素之后的最大值
        for(int j = i + 1; j < n; j++) {
            if (x[j] > x[k]) {
                k = j;
            }
        }
        //若后续有比x[i]大的元素，交换
        if(k != i) {
            t = x[i];
            x[i] = x[k];
            x[k] = t;
        }
    }
}
```

指针作为函数参数：与数组作函数参数的异同

如果有一个实参数组，想在函数中改变此数组中的元素的值，有四种方式

实参--数组
形参--数组

```
int  main(void)
{
        int a[10];
        f(a, 10);
}

void f(int x[], int n)
{
        ...
}
```

实参--数组
形参--指针

```
int  main(void)
{
        int a[10];
        f(a, 10);
}

void f(int* x, int n)
{
        ...
}
```

实参—指针
形参--数组

```
int  main(void)
{
        int a[10],
        *p=a;
        f(p, 10);
}
void f(int x[], int n)
{
        ...
}
```

实参--数组
形参--指针

```
int  main(void)
{
        int a[10],
        *p=a;
        f(a, 10);
}
void f(int* x, int n)
{
        ...
}
```

指针作为函数参数：const限定

指针变量做函数参数时

- 在函数内部就有可能通过指针间接修改指向**对象的值**，为避免此类操作可以对指针参数进行const限定

例如：

```
void fun1(const char* p, char m)
{
    *p = m; //错误，*p是只读的
    p = &m; //正确，指针变量p可以修改
}
```

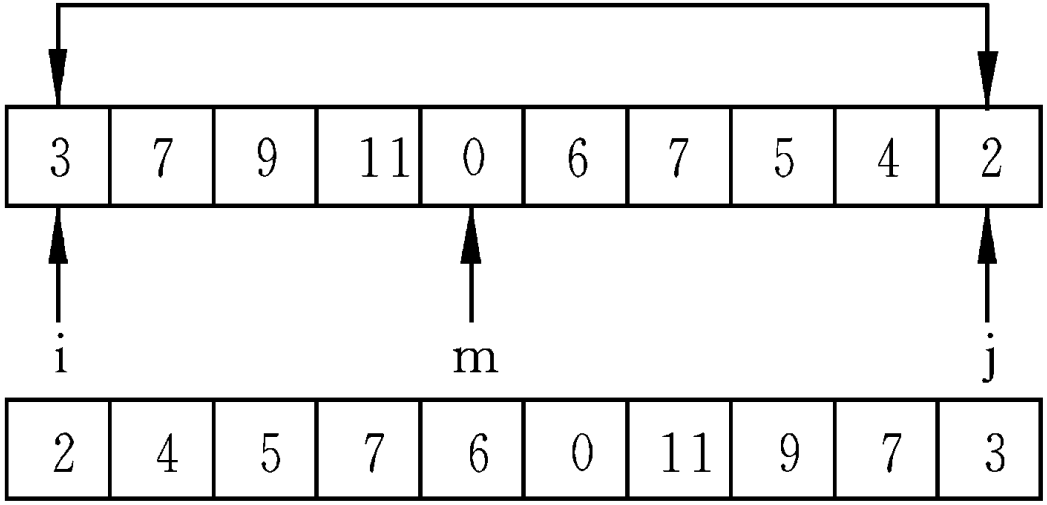
- 如果不允许在函数内部修改形参**指针变量的值**，可对指针变量进行const限定

例如：

```
void fun2(char* const p, char m)
{
    *p=m; //正确*p是可以修改的
    p=&m; //错误指针变量p是只读的
}
```

指针作为函数参数：实例

将数组 a 中 n 个整数按相反顺序存放。



指针作为函数参数：一级指针实例

```
void inv(int x[], int n) {
    int temp;
    int m = (n - 1) / 2;
    for(int i = 0; i <= m; i++) {
        temp = x[i];
        x[i] = x[n – 1 - i];
        x[n – 1 - i] = temp;
    }
}
```

```
void inv(int* x, int n) {
    int temp
    int* i = x;
    int* j = x + n - 1;
    int m = (n - 1) / 2;
    int* p = x + m;
    for(; i <= p; i++, j--) {
        temp = *i;
        *i = *j;
        *j = temp;
    }
}
```

指针作为函数参数：二级指针实例

假设主调函数中有定义

```
int a = 10;
int b = 20;
int* p1 = &a;
int* p2 = &b;
```

如果一个函数fun的功能是将两个指针的值交换，即函数调用后p1指向b，p2指向a，那么如何设计该函数？

- 传递一级指针时，对指针指向的内存变量做有效操作；
- 传递二级指针时，对指针的指向做有效改变；

指针作为函数参数：二级指针实例

若要在函数fun中**修改p1和p2的值**，函数调用就必须用p1和p2的地址作为实参，即

```
fun(&p1, &p2);
```

&p1的类型应是二级指针，因此函数fun应如下定义

```
void fun(int** x, int** y) { //指向指针的指针变量作为函数形参
    int *t; //指针类型
    t = *x;
    *x = *y;
    *y = t; //*x和*y为指针类型，两个指针交换
}
```

指针作为函数参数：二级指针实例

将二维数组进行按行逆序

```
int A[4][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12},
    {13, 14, 15, 16}
};
```

```
int i = 0;
int j = 3;
while (i < j) {
    swaprow(A + i, A + j); //交换A+i和A+j行的元素
    i++;
    j--;
}
```

函数形参可以是指向数组的指针变量，例如：

```
void swaprow(int (*p1)[4], int (*p2)[4]) { //交换p1和p2指向的一维数组的元素
    int i;
    int t;
    for (i = 0; i < 4; i++) {
        t = *(*p1 + i), *(*p1 + i) = *(*p2 + i), *(*p2 + i) = t;
    }
} //这种方法太费劲了，实际应用中直接用数组做参数
```

指针作为函数参数：字符串

类似数字型数组，字符串作为函数参数传递的是地址，被调函数的修改会返回到主调函数

函数形参可用字符数组，也可用指针变量，两种形式等价

~~实际编程中，程序员更偏爱用字符指针变量作为函数形参~~

```
char* strcpy(char* s1, const char* s2); //字符串复制函数
char* strcat(char* s1, const char* s2); //字符串连接函数
int strcmp(const char* s1, const char* s2); //字符串比较函数
int strlen(const char* s); //计算字符串长度函数
```

本节课主要内容

指针作为函数参数

指针作为函数返回类型

函数指针

动态内存管理

指针小结

返回指针值的函数

一个函数可以带回一个整型值、字符值、实型值等，也可以带回指针型的数据，即地址。

返回指针值的函数，定义形式为

```
类型名* 函数名（参数列表）
{
        函数体
}
```

例如：

```
int* a(int  x, int y) { …}

char* substring(const char *str, const char *sub) {…}
```

返回指针值的函数

函数返回指针值，需要考虑**指针有效性**的问题

```
char* substring(const char* str, const char* sub)
{
    char a='A';
    return &a; //正确返回值&a与返回类型char *匹配，但“无效”了
}
```

函数调用结束后，局部变量a被释放，主调函数获得的指针无效。

一般地，函数应返回（比返回值生命周期更长的实体指针）：

- 由主调函数传递进去的有效指针值；
- 动态分配得到的指针值（后面将要讲到）；
- 0值指针，表示无效指针。

返回指针值的函数：实例

有若干个学生的成绩（每个学生有 4 门课程），要求在用户输入学生序号以后，能

```
#include <stdio.h>
float* search(float (*pointer)[4], int n) {
    float* pt = *(pointer + n);
    return pt;
}



int main(void) {
    float score[][4] = {
        {60, 70, 80, 90},
        {56, 89, 67, 88},
        {34, 78, 90, 66}
    };
    float* p;
    int m;
    printf("enter the number of  student:");
    scanf("%d", &m);
    p = search(score, m);
    for(int i = 0; i < 4; i++) {
        printf("%5.2f\t", *(p + i));
    }
    return 0;
}
```

函数指针

函数代码在内存中也要占据一段存储空间（代码区），这段存储空间的起始地址称为函数入口地址。

C语言规定函数入口地址为**函数的指针**，即**函数名**
既代表函数，又是函数的指针。

指向函数的指针变量，定义形式为：

返回类型 **(*函数指针变量名)**(形式参数列表), ……;

注意：括号不能省略

它指向如下函数

```
返回类型 函数名(形式参数列表)
{
}
```

函数指针：赋值与引用操作

与数据对象的指针不同，函数指针一般只有赋值和间接引用的操作

赋值：将函数的地址赋值给函数指针变量

```
int max(int a, int b); //max函数原型
int min(int a, int b); //min函数原型

typedef int FunType(int, int);//定义函数类型

int (*p)(int a, int b); //定义函数指针变量
FunType p1;//定义函数指针变量
```

函数指针变量与指向函数**具有相同的函数原型**

函数指针：赋值与引用操作

引用：通过函数指针调用函数

```
int max(int a, int b); //max函数原型
int min(int a, int b); //min函数原型


typedef int FunType(int, int);//定义函数类型

int (*p)(int a, int b); //定义函数指针变量
……
p = max;
c = p(a, b); // 也可以写作c=(*p)(a,b), 等价于c=max(a,b);
p = min;
c = p(a, b); // 也可以写作c=(*p)(a,b), 等价于c=min(a,b);
```

函数指针：应用实例

指向函数的指针多用于指向不同的函数，实现动态调用

常见用途之一是**把指针作为参数传递到其他函数。**
广泛用于菜单设计、事件驱动、动态链接库等场合

```c
#include <stdio.h>
int max(int x, int y) {
    return (x > y) ? x : y;
}

int min(int x, int y) {
    return (x<y)?x:y;
}

int add(int x, int y) {
    return(x+y);
}
//typedef int FunType(int, int);
//void process(int x, int y, FunType* fun)
void process(int x, int y, int (*fun)(int,int)) {
    printf("%d\n", fun(x, y));
}
```

```c
int main(void) {
    int a;
    int b;
    scanf("%d, %d", &a, &b);
    printf("max =");
    process(a, b, max);
    printf("min =");
    process(a, b, min);
    printf("sum =");
    process(a, b, add);
}
```

动态内存管理

静态内存：编译时为程序中的数据对象分配相应存储空间

- 编译时空间大小必须明确，数组的长度必须是常量
- 运行期间不可以更改大小
- 全局变量、局部变量、静态变量等
- 在数据区分配空间

动态内存：程序运行中根据需要动态地申请或释放内存

- 编译阶段不需要预先分配存储空间
- 运行时根据程序需要适时、适量分配，并可扩大或缩小空间
- 在堆（heap）分配内存，空间上限是物理内存的上限
- 分配和释放开销大

动态内存管理：基本函数

C语言动态内存管理是通过标准库函数来实现的

```
#include <stdlib.h>
```

动态内存分配函数（1）：malloc　　memory allocate

```
void* malloc(size_t size);
```

- 参数size：申请分配的字节数，类型size_t一般为unsigned int
- 返回值：若分配成功，函数返回一个指向该内存空间起始地址的void指针；分配失败（如空间不足），函数返回0值指针NULL。

注意：

- 对返回值进行检查
- malloc对分配得的内存空间并未初始化，使用前务必进行初始化
- 返回的void指针可以显式转换为其他指针类型
- 不要丢失起始地址，否则内存泄露

动态内存管理：基本函数

动态内存分配函数（2）：calloc

complex allocate

```
void* calloc(size_t nmemb, size_t size);
```

- 功能：分配nmemb个连续的指定大小的内存空间
- 参数：nmemb个内存空间block，每个内存空间block的大小为size个字节，总字节为nmemb*size
- 返回值：若分配成功，函数返回一个指向该内存空间起始地址的void类型指针，**并初始化为0**；分配失败，返回0值指针NULL。

```
int* p;
p = (int*)calloc(50, sizeof(int)); //初始化
p = (int*)malloc(50*sizeof(int)); //未初始化
```

动态内存管理：基本函数

动态内存调整函数：realloc　　re-allocate

```
void* realloc(void* ptr, size_t size);
```

- 功能与参数：将指针ptr所指向的动态内存空间扩大或缩小为size大小，
  扩大空间时原有内存中的内容保持不变，缩小空间可能会丢失缩小的部分内容。
- 返回值：如果调整成功，函数返回一个指向调整后的内存空间起始地址的void类型指针

```
int *p;
p = (int*)malloc(50*sizeof(int) ); //分配一个有50个int整型的内存空间
p = (int*)realloc(p, 10*sizeof(int) ); //调整为有10个int整型的内存空间
p = (int*)realloc(p, 100*sizeof(int) ); //再次调整为有100个整型的内存空间
```

动态内存管理：基本函数

动态内存释放函数： free

```
void free(void *ptr);
```

- 功能：释放ptr指向的动态分配的内存空间，如果ptr为NULL则什么也不做
- 参数：ptr指向已有的动态内存空间

注意：

- ptr释放之后，ptr的内容（所存放的地址值）并不会改变
- 因此，free操作后需要设置ptr等于NULL，避免产生“无效指针”

动态内存管理：基本函数

动态内存批量赋值：memset

```
void* memset(void *s, int c, size_t n);
```

功能及参数：将已开辟内存空间 s 的首 n 个字节的值设为值 c

例如：

```
char str[100];
memset(str, 0, 100);
char a[100];
memset(a, '/0', sizeof(a));
```

内存拷贝：memcpy

```
void* memcpy(void* dest, void* src, unsigned int count);
```

功能及参数：从src所指向的内存区域 拷贝 count个字节的内容 到dest所指向的内存区域

动态内存管理：注意事项

动态内存管理按程序员人为的指令进行，生命期由程序员决定，允许跨越函数。

动态内存分配和释放必须对应（
遵守实体生存周期必须不小于指针变量生存周期的原则）

- 不释放内存会产生“内存泄漏”
- 再次释放已经释放的内存空间，会导致程序崩溃性

动态分配内存一般需要**人为的指令赋初始值**

避免释放内存后出现“无效指针”，应及时设置为空指针

动态内存：实例

```c
#include <stdio.h>
#include <stdlib.h>
//根据行、列动态生成一个二维数组
int**  CreatArray(int m1, int n1);
//根据行、列动态销毁一个二维数组
void DestroyArray(int** arr, int m1, int n1);
int main(void) {
    //2级指针代表2维数组
    int** a;
    //2维数组的行列维度
    int n;
    int m;
    printf("Please input m and n:");
    //主函数输入二维数组的维度
    scanf("%d %d", &m, &n);
    //函数调用生成一个动态二维数组a
    a = CreatArray(m, n);
    //引用动态数组的元素，进行赋值
    for(int i = 0; i < m; i++)  {
        for(int j = 0; j < n; j++) {
            // *(*(a+i)+j))=i+j;
            a[i][j] = i + j;
        }
    }
    //释放空间的语句或者函数
    DestroyArray(a, m,  n);
    return 0;
}
```

```c
int** CreatArray(int m1, int n1) {
    int** p;
    p = (int**)malloc(sizeof(int*) * m1);
    for(int i = 0; i < m1; i++)
        *(p + i) = (int*)malloc(sizeof(int) * n1);
    return p;
}
```

```c
void DestroyArray(int** arr, int m1, int n1)
{
    for(int i = 0; i < m1; i++) {
        free(*(arr + i));
    }
    free(arr);
}
```

动态内存管理：实例

跨函数内存申请：参数传递

```
void GetMemory(char* p, int num) {
    //这里函数调用结束后内存丢失
    p = (char*)malloc(sizeof(char) * num);
}


void Test1(void) {
    char* str = NULL;
    //str仍然为NULL
    GetMemory(str, 50);
    // 运行错误
    strcpy(str, "hello");
}
```

深刻理解函数参数传递的本质

动态内存管理：实例

跨函数内存申请：参数传递

```
void GetMemory2(char** p, int num){
    *p = (char*)malloc(sizeof(char) * num);
}


void Test2(void) {
    char* str = NULL;
    GetMemory2(&str, 50); //参数是&str
    strcpy(str, "hello");
    printf("%s", str);
    free(str);
}
```

深刻理解函数参数传递的本质

动态内存管理：实例

跨函数内存申请：返回值

```
char* GetString(void) {
    char p[] = "hello world";
    return p;  // 编译器将提出警告
}


void Test4(void) {
    char* str = NULL;
    // str所指向空间的生命周期小于str自己的生命周期，错误
    str = GetString();
    printf("%s", str);
}
```

切忌传递来自栈的内存空间！！！

动态内存管理：实例

跨函数内存申请：返回值

```
char* GetMemory3(int num) {
    char* p = (char*)malloc(sizeof(char) * num);
    return p;
}


void Test3(void) {
    char* str = NULL;
    str = GetMemory3(100);
    memset(str, 0, 100 * sizeof(int));
    strcpy(str, "hello");
    printf("%s", str);
    free(str);
}
```

切忌传递来自栈的内存空间！！！

指针小结：定义

定义	含义
int i;	定义整型变量i
int* p;	p为指向整型数据的指针变量
int a[n];	定义整型数组a，它有n个元素
int* p[n];	定义指针数组p，它由n个指向整型数据的指针元素组成
int (*p)[n];	p为指向含n个元素的一维数组的指针变量
int f();	f为带回整型函数值的函数
int* p();	p为返回一个指针值的函数，该指针指向整型数据
int (*p)();	p为指向函数的指针，该函数返回一个整型值

指针小结：运算

算数运算

- 指针变量加（减）一个整数、自增自减

- 指针相减，计算指针间元素个数

比较运算

强制类型转换

- 编译器在赋值、参数传递、算术运算时不做自动类型转换
- 特定需要时可以进行强制类型转换，产生目标类型的临时指针

指针小结：赋值

赋值时务必保证指向类型一致、指针级次一致

例如：

```
int a, array[10], matrix[3][4], *p, *p1, *p2, *pp;
p = &a;  //将变量 a 的地址赋给 p
p=array;  //将数组array首元素地址赋给p
p=&array[i];//将数组array第i个元素的地址赋给p
p1=p2; //p1和p2都是指针变量，将p2的值赋给p1
pp = matrix; //将数组matrix首行地址赋给pp
```

```
int max(int x, int y, float z);
int (*p) (int x, int y, float z);
p=max; //max为已定义的函数，将max的入口地址赋给p
```

指针小结：状态

指向一个已知对象

例如： int a;
int* p = &a;

指向空地址，即零地址或NULL

- 用作初始化或操作前判断

例如：

```
int* p = NULL;
if (p != NULL) …
```

指向未知对象

- 野指针：未进行初始化
- 迷途指针：指向对象内存释放或者访问数组越界

指针小结：void指针

可进行强制类型转换使之适合于被赋值的变量的类型

例如：

```
char* p1;
void* p2;
p1 = (char*)p2;
```

可以不做类型转换而指向任何类型

例如：

```
void* pv1;
pv1 = &x; //无需指针类型转换
```

不可做算数运算、可做比较运算

指针小结：指针与函数

指针作为函数参数，与数组作函数参数类似

- 形参与实参传地址值、主调函数与被调函数共享内存，函数调用结束后操作结果保留
- 可以返回多个结果
- 不检查越界操作，需要传送数据大小

指针作为函数返回值

- 一旦使用，要特别注意指针的有效性

函数指针

- 一般用于根据用户需求选择特定函数的情形