# 第一章 指针和数组

## 第一部分：基本知识

### 一、指针

计算机中的内存都是[编址](https://baike.baidu.com/item/%E7%BC%96%E5%9D%80" \t "_blank)的，就像你家的地址一样。在[程序编译](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E7%BC%96%E8%AF%91" \t "_blank)或者运行的时候，系统（可以不关心具体是什么，可能是[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)，也可能是操作系统）开辟了一张表。每遇到一次声明[语句](https://baike.baidu.com/item/%E8%AF%AD%E5%8F%A5" \t "_blank)（包括函数的传入参数的声明）都会开辟一个内存空间，并在表中增加一行纪录。记载着一些对应关系。（如图所示）

int np;

char myChar;

int \*myPointer;

char \*myPointer2;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Address | Length |  |
| 1 | np | 2000 | 4B |  |
| 2 | myChar | 2004 | 1B |  |
| 3 | myPoint | 2005 | 4B |  |
| 4 | myPoint2 | 2009 | 4B |  |

指针，是一种保存变量地址的变量，也是一个[无符号整数](https://baike.baidu.com/item/%E6%97%A0%E7%AC%A6%E5%8F%B7%E6%95%B4%E6%95%B0" \t "_blank)（unsigned int），它是一个以当前系统寻址范围为取值范围的整数。32位系统下寻址能力（[地址空间](https://baike.baidu.com/item/%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4" \t "_blank)）是4G Bytes（0~2^32-1）[二进制](https://baike.baidu.com/item/%E4%BA%8C%E8%BF%9B%E5%88%B6" \t "_blank)表示长度为32bits（也就是4Bytes）， unsigned int类型也正好如此取值。

寻址方法：

直接寻址是一种基本的寻址方法，其特点是：在指令格式的地址的字段中直接指出操作数在内存的地址。由于操作数的地址直接给出而不需要经过某种变换，所以称这种寻址方式为直接寻址方式。在指令中直接给出参与运算的操作数及运算结果所存放的主存地址，即在指令中直接给出有效地址，例如:

Short int a=10;

&a=0023FF74

|  |
| --- |
| 变量a的低位字节 |
| 变量a的高位字节 |

这是直接按变量名来存取变量值的访问方式。

间接寻址是在[直接寻址](https://baike.baidu.com/item/%E7%9B%B4%E6%8E%A5%E5%AF%BB%E5%9D%80" \t "_blank)的基础上面建立起来的，也就是直接寻址得到的数据是一个地址，通过这个地址找到最终的数据，也就是两次寻址，第一次得到的是地址，第二次才是目标数据。

例如：

Short int a=10;

Short int \*p=&a;

|  |
| --- |
| 变量a的低位字节 |
| 变量a的高位字节 |
| 。。。 |
| 变量a的地址&a |

通过指针变量访问它指向的变量的值。C语言提供了&和\*，“&”用于得到变量的地址，“\*”用来得到指针所指对象的内容，“\*”称为指针运算符，也称为间接寻址运算符。

指针之所以重要，原因有四点：

指针为函数提供修改变量值的手段

指针为C语言的内存动态分配系统提供支持

指针为动态数据结构（如链表、队列、二叉树等）提供支持

指针可以改善子程序的效率。

### 二、数组

1. 数组的定义

所谓数组，是无序的元素序列。 若将有限个类型相同的变量的集合命名，那么这个名称为数组名。组成数组的各个变量称为数组的分量，也称为数组的元素，有时也称为下标变量。用于区分数组的各个元素的数字编号称为下标。数组是在程序设计中，为了处理方便， 把具有相同类型的若干元素按无序的形式组织起来的一种形式。 这些无序排列的同类数据元素的集合称为数组。

数组是用于储存多个相同类型数据的集合。

2.数组的引用

数组的引用方式为：

数组名[下标1][下标2]…[下标N]

例如：

For(i=0;i<3;i++) //行下标值变化

{

For(j=0;j<4;j++)//列下标值变化

{

Scanf(“%d”,&matrix[i][j]);

}

}

3. 数组的初始化

（1）在定义数组时对数组元素赋以初值。如:

static int a［10］=｛0，1，2，3，4，5，6，7，8，9｝;

经过上面的定义和初始化后，a［0］=0，a［1］=1，…，a［9］=9。

（2）初始化时可以只对一部分元素赋初值。例如:

static int a［10］=｛0，1，2，3，4｝;

定义的数组有10个元素，但只对其中前5个元素赋了初值，后5个元素初值为0。

（3）如果想使一个数组的元素值全部为0，可以用下面的方法:

|  |  |
| --- | --- |
|  | static int a［10］=｛0，0，0，0，0，0，0，0，0，0｝; |

不能用:

|  |  |
| --- | --- |
|  | static int a［10］=｛0\*10｝; |

如果对static型数组不赋初值，系统会对定义的所有数组元素自动赋以0值。

（4）在对全部数组元素赋初值时，可以不指定数组长度。

### 三、指针和数组之间的关系

1.数组并不是指针

    首先，对数组的引用总是可以写成对指针的引用，这是数组极其普遍的用法。但并非所有情况都可以这样。

    访问方式：

     编译器为每个变量分配一个地址，该地址在编译时可知，存储于变量中的值，在运行时可知，数组是不可修改的左值，其地址在编译时可知，因此不需要增加指令取地址，但对于指针，首先必须在运行时取得其当前值，然后才能对其解除引用操作。

    对于数组元素的引用，只要获取数组地址，加上相应的偏移即可。对于指针，首先要在根据相应符号的地址取得该地址处的内容，再取内容所指的内容。也就是说，数组的地址，在编译器中一直可知，但指针的地址，需要编译器取得符号表P的地址，提取存储于此处的指针。

    字符串常量初始化：

两者都可以，但实现机制不一样。

      定义指针时，编译器只是为该指针分配地址，除非在定义同时给该指针一个字符串常量初始化，否则不会为所指向的对象分配空间。且只有字符串常量才可以这样。在ANSI C中，初始化指针时所创建的字符串常量被定义为只读，如果试图用指针修改这个字符串的值，就会出现未定义的行为，在有些编译器中，字符串常量被放在只允许读取的文本段，以防止被修改。

数组也可以用字符串常量初始化，其中字符可以在后期修改。

对于编译器而言，一个数组就是一个地址，一个指针就是一个地址的地址。

2.数组和指针可以互换

C语言标准要求编译器必须具备对数组的引用在编译时改写成指针的形式的能力。编译器可以自动计算偏移的实际步长，这就是为什么指针总是有类型限制。

有一种说法，在编写数组算法时，使用指针比使用数组更有效率，这个颇为人们所接受的说法在通常情况下是错误的，在现代的产品质量优化的编译器，一维数组和指针引用所产生的代码并不具有显著的差别，数组的下标是定义在指针基础上的，所以优化器常常把它转换为更有效率的指针表达式，并生成相同的机器指令。

     实际上，如果一个经过良好优化的编译器进行代码分析，并把基本变量放在高速寄存器中来确认循环是否继续，那么最终在循环访问指针和数组所产生的代码很可能是相同的，在处理一维数组时，指针并不见的比数组快，C语言把数组下标改写成指针偏移量的根本原因是指针和偏移量是底层硬件所使用的基本模型。

    对于数组和指针相同的地方，只是在表达式中，数组和指针可以互换。因为他们最终都被编译器改写为指针形式，都可以进行取下标操作。就像加法一样，取下标操作符的操作数顺序是可以互换，即a[6]和6[a]都可以取同一个内容。只是6[a]这种形式不用而已。

   另外，作为新参传的数组名和指针也是等价的，这主要是处于效率的考虑。

3.多维数组

    实际上，C语言只支持数组的数组，术语上称为多维数组。

指向字符串的一维指针数组

可以通过声明一个一维指针数组，其中每个指针指向一个字符串来取得类似二维字符数组的效果。例如：char \*array[4];

用于实现多维数组的指针数组有多种名字，如Iliffe向量，display，dope向量。这种数组必须用指向为字符串而分配的内存的指针进行初始化，可以为每个指针分配内存，也可以一次性分配，如p[i][j],一次性分配方法：

malloc（x \* y \* sizeof(T)）,然后使用一个循环，用指针指向这块内存的各个区域，整个数组能够保证在连续的内存中，**这种方法减少了malloc的开销，但无法单独释放一个字符串。**

   Iliffe向量，用来处理存储各行长度不一样的表以及一个函数调用中传递一个字符数组，如果声明一个字符串指针数组，并根据字符串的长度分配内存，将大大节省系统资源，有人把这个数组称为锯齿状数组。例如：

char \*turnip[10];

char my\_string[] = "hello";

turnip[i] = &my\_string[0];

但是，Iliffe向量可能会使分配的字符串在不同的页面中，这就违反了局部引用的原则，并导致频繁的页面交换，具体如何取决于怎样访问数据以及访问的频度。

数组名被改写成一个指针参数，并不是一个递归定义，即数组的数组被修改为数组的指针，而不是指针的指针。

在C语言中没有办法向函数传递一个普通的多维数组。因为我们需要知道每一维的长度，以便为地址运算提供正确的单位长度，在C语言中我们没有办法在实参和形参之间交流这种数据，因为它在每次调用时会改变。因此必须提供除了最左边一维以外的所有维长度。也就是说，你定义的函数只能为：

T fun(T array[][2][2]);这样除了最左侧的一维其他长度固定的特殊多维数组，进行如下调用：

T arrayTmp[2][2][2];

fun(arrayTmp);

如果除最左边的一维其他维长度与形参不一样，编译不通过。

但上面的方法不能满足一般情况，因此，将多维数组传递给函数，有以下几种：

方法1.

直接传递，my\_fun(T my\_array[2][2]);即只能传递特殊的多维数组。

方法2.

省略一维长度，

my\_fun(T my\_array[][2]);也可以my\_fun(T （\*my\_array）[2]);

方法3.

放弃二维数组，将其构建成Iliffe向量，my\_fun(T \*\*my\_array);

注意：只有把二维数组改为一个指向向量的指针数组的前提下才可以这样。

因为字符串和指针都有一个显式的边界NUL和NULL，可以作为结束标记。因此，必须是指针数组，必须是指向字符串的指针数组。

### 四、指针数组和数组指针

指针数组：array of pointers，即用于存储指针的数组，也就是数组元素都是指针

数组指针：a pointer to an array，即指向数组的指针

int\* a[4]     指针数组

                 表示：数组a中的元素都为int型指针

                 元素表示：\*a[i]   \*(a[i])是一样的，因为[]优先级高于\*

int (\*a)[4]   数组指针

                 表示：指向数组a的指针

                 元素表示：(\*a)[i]

typedef int\* pInt;

pInt a[4];

这跟上面指针数组定义所表达的意思是一样的，只不过采取了类型变换。

代码演示如下：

#include <iostream>

using namespace std;

int main()

{

int c[4]={1,2,3,4};

int \*a[4]; //指针数组

int (\*b)[4]; //数组指针

b=&c;

//将数组c中元素赋给数组a

for(int i=0;i<4;i++)

{

a[i]=&c[i];

}

//输出看下结果

cout<<\*a[1]<<endl; //输出2就对

cout<<(\*b)[2]<<endl; //输出3就对

return 0;

}

定义了数组指针，该指针指向这个数组的首地址，必须给指针指定一个地址，容易犯的错得就是，不给b地址，直接用(\*b)[i]=c[i]给数组b中元素赋值，这时数组指针不知道指向哪里，调试时可能没错，但运行时肯定出现问题，使用指针时要注意这个问题。但为什么a就不用给他地址呢，a的元素是指针，实际上for循环内已经给数组a中元素指定地址了。但若在for循环内写\*a[i]=c[i]，这同样会出问题。

### 六、函数指针

函数指针是指向函数的指针[变量](https://baike.baidu.com/item/%E5%8F%98%E9%87%8F/3956968" \t "_blank)。 因此“函数指针”本身首先应是指针变量，只不过该指针变量指向函数。这正如用指针变量可指向整型变量、字符型、数组一样，这里是指向函数。如前所述，C在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，在这些概念上是大体一致的。函数指针有两个用途：调用函数和做函数的参数。

函数指针的声明方法为：

返回值类型 ( \* [指针变量](https://baike.baidu.com/item/%E6%8C%87%E9%92%88%E5%8F%98%E9%87%8F" \t "_blank)名) ([[形参](https://baike.baidu.com/item/%E5%BD%A2%E5%8F%82" \t "_blank)列表]);

注1：“返回值类型”说明函数的返回类型，“(指针变量名 )”中的括号不能省，括号改变了运算符的优先级。若省略整体则成为一个函数说明，说明了一个返回的数据类型是指针的函数，后面的“形参列表”表示指针变量指向的函数所带的参数列表。例如：

int func(int x); /\* 声明一个函数 \*/

int (\*f) (int x); /\* 声明一个函数指针 \*/

f=func; /\* 将func函数的首地址赋给指针f \*/

或者使用下面的方法将函数地址赋给函数指针：

f = &func;

赋值时函数func不带括号，也不带参数，由于func代表函数的首地址，因此经过赋值以后，指针f就指向函数func(x)的代码的首地址。

函数括号中的[形参](https://baike.baidu.com/item/%E5%BD%A2%E5%8F%82" \t "_blank)可有可无，视情况而定。

下面的程序说明了函数指针调用函数的方法：

#include<stdio.h>

int max(int x,int y){return (x>y? x:y);}

int main()

{

    int (\*ptr)(int, int);

    int a, b, c;

    ptr = max;

    scanf("%d%d", &a, &b);

    c = (\*ptr)(a,b);

    printf("a=%d, b=%d, max=%d", a, b, c);

    return 0;

}

ptr是指向函数的[指针变量](https://baike.baidu.com/item/%E6%8C%87%E9%92%88%E5%8F%98%E9%87%8F" \t "_blank)，所以可把函数max()赋给ptr作为ptr的值，即把max()的入口地址赋给ptr,以后就可以用ptr来调用该函数，实际上ptr和max都指向同一个入口地址，不同就是ptr是一个指针变量，不像函数名称那样是死的，它可以指向任何函数，就看你想怎么做了。在程序中把哪个函数的地址赋给它，它就指向哪个函数。而后用指针变量调用它，因此可以先后指向不同的函数。不过注意，指向函数的[指针变量](https://baike.baidu.com/item/%E6%8C%87%E9%92%88%E5%8F%98%E9%87%8F" \t "_blank)没有++和--运算，用时要小心。

不过，在某些[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)中这是不能通过的。这个例子的补充如下。

应该是这样的：

1.定义函数指针类型：

typedef int (\*fun\_ptr)(int,int);

2.声明变量，赋值：

fun\_ptr max\_func=max;

也就是说，赋给函数指针的函数应该和函数指针所指的函数原型是一致的

#include<stdio.h>

void FileFunc()

{

printf("FileFunc\n");

}

void EditFunc()

{

printf("EditFunc\n");

}

void main()

{

typedef void(\*funcp)();

funcp pfun=FileFunc;

pfun();

pfun=EditFunc;

pfun();

}

指针数组

int (\*result（）)[20]

七、复杂声明

在分析复杂声明时，要借助typedef把复杂声明分解成几种基本形式：

* T \*p;，p是指向T类型的指针。
* T a[];，a是由T类型的元素组成的数组，但有一个例外，如果a是函数的形参，则相当于T \*a;
* T1 f(T2, T3...);，f是一个函数，参数类型是T2、T3等等，返回值类型是T1。

我们分解一下这个复杂声明：

int (\*(\*fp)(void \*))[10];

1、fp和\*号括在一起，说明fp是一个指针，指向T1类型：

typedef int (\*T1(void \*))[10];

T1 \*fp;

2、T1应该是一个函数类型，参数是void \*，返回值是T2类型：

typedef int (\*T2)[10];

typedef T2 T1(void \*);

T1 \*fp;

3、T2和\*号括在一起，应该也是个指针，指向T3类型：

typedef int T3[10];

typedef T3 \*T2;

typedef T2 T1(void \*);

T1 \*fp;

显然，T3是一个int数组，由10个元素组成。分解完毕。

## 第二部分：实训任务

### 实现函数strcmp,strncmp

写函数，用指针实现两个字符串的比较。即自己写一个strcmp、strncmp函数

函数原型为：

int strcmp(char \*p1,char \*p2)

int strncmp(char \*p1,char \*p2,int n)

设p1指向字符串s1，p2指向字符串s2。要求：当s1=s2时，返回值为0。当s1不等于s2时，返回它们二者的第一个不同字符的ASCII码差值（如“BOY”与“BAD”，第二字母不同，“O”与“A”之差为79-65=14）；如果s1>s2，则输出正值；如果s1<s2，则输出负值。

**strcmp:**

int strcmp(char \*p1, char \*p2)

{

    while (\*p1 != '\0' && \*p2 != '\0')

    {

        if (\*p1 != \*p2)

        {

            return \*p1 - \*p2;

        }

        p1++;

        p2++;

    }

    // 前缀相同且长度相同

    if (\*p1 == '\0' && \*p2 == '\0')

    {

        return 0;

    }

    else

    {

        // 前缀相同，长度不同

        if (\*p2 == '\0')

        {

            return 1;

        }

        else

        {

            return -1;

        }

    }

}

**strncmp:**

int strncmp(char \*s1, char \*s2, int n)

{

    int cnt = 0;

    while (\*s1 != '\0' && \*s2 != '\0' && cnt < n)

    {

        if (\*s1 != \*s2)

        {

            return \*s1 - \*s2;

        }

        s1++;

        s2++;

        cnt++;

    }

    if (\*s1 == '\0' && \*s2 == '\0')

    {

        return 0;

    }

    else

    {

        // 前缀相同，长度不同

        if (\*s2 == '\0')

        {

            return 1;

        }

        else

        {

            return -1;

        }

    }

}

### 升级strcat，strncat

写函数，用指针实现两个字符串的拼接。即自己写一个strcat、strncat函数，但它不会溢出目标数组。复制结果必须是一个真正的字符串。

函数原型为：

char\* strcat(char \*dest,const char \*src);

char\* strncat(char \*dest,const char \*src,size\_t n);

**strcat**

void strcat(char \*dest, char \*src)

{

const int MaxLen = 10;

    int len1 = 0, len2 = 0;

    while (dest[len1] != '\0' && len1 < 2 \* MaxLen)

    {

        len1++;

    }

    while (src[len2] != '\0' && len2 < MaxLen)

    {

        len2++;

    }

    int cnt = 0;

    while ((\*dest) != '\0')

    {

        dest++;

        cnt++;

    }

    // 考虑终止符

    while ((\*src) != '\0' && cnt < MaxLen - 1)

    {

        \*dest = \*src;

        dest++;

        src++;

        cnt++;

    }

    // 加上终止符

    \*dest = '\0';

}

**strncat**

void strncat(char \*dest, char \*src, int n)

{

const int MaxLen = 10;

    // cnt1统计不超过n

    int cnt1 = 0;

    int len1 = 0, len2 = 0;

    while (dest[len1] != '\0' && len1 < 2 \* MaxLen)

    {

        len1++;

    }

    while (src[len2] != '\0' && len2 < MaxLen)

    {

        len2++;

    }

    // cnt2使不越界

    int cnt2 = 0;

    while ((\*dest) != '\0')

    {

        dest++;

        cnt2++;

    }

    // 考虑终止符

    while ((\*src) != '\0' && cnt2 < MaxLen - 1 && cnt1 < n)

    {

        \*dest = \*src;

        dest++;

        src++;

        cnt1++;

        cnt2++;

    }

    // 加上终止符

    \*dest = '\0';

}

### 升级my\_strcpy

编写一个名叫my\_ strcpy的函数。它类似于strcpy函数，但它不会溢出目标数组。复制结果必须是一个真正的字符串。

void my\_strcpy(char \*dest, char \*src)

{

const int MaxLen = 10;

    int len = 0;

    while (\*src != '\0' && len < MaxLen - 1)

    {

        \*(dest + len) = \*src;

        len++;

        src++;

    }

    \*(dest + len) = '\0'; // 添加终止符

}

### my\_strcpy\_end

编写一个名叫my\_strcpy\_end的函数取代strcpy函数，它返回一个指向目标字符串末尾的指针（指向MUL字节的指针），而不是返回一个指向目标字符串起始位置的指针。

char \*my\_strcpy\_end(char \*dest, char \*src)

{

const int MaxLen = 10;

int len = 0;

    while (\*src != '\0' && len < MaxLen - 1)

    {

        \*dest = \*src;

        dest++;

        src++;

        len++;

    }

    // 添加终止符

    \*dest = '\0';

    return dest;

}

### 插入子串

设计函数char \*insert(s1,s2,n),用指针实现在字符串s1中的指定位置n处插入字符串s2

char \*insert(const char \*s1, const char \*s2, int n)

{

    int len1 = strlen(s1);

    int len2 = strlen(s2);

    int len = len1 + len2;

    // 为新字符串分配内存

    char \*result = (char \*)malloc((len + 1) \* sizeof(char));

    // 复制 s1 的前半部分到新字符串

    strncpy(result, s1, n);

    // 插入 s2

    strcpy(result + n, s2);

    // 复制 s1 的后半部分到新字符串

    strcpy(result + n + len2, s1 + n);

    return result;

}

### 实现atoi

采用指针的方式实现atoi函数

int atoi(char \*s)

int atoi(char \*s)

{

    int res = 0;

    while (\*s != '\0')

    {

        res = res \* 10 + (\*s - '0');

        s++;

    }

    return res;

}

### 逆序排列

将n个数按输入时顺序的逆序排列,用函数实现

void reverse(int a[ ],int n)

void reverse(int a[], int n)

{

    for (int i = 0; i < n / 2; i++)

    {

        int temp = a[i];

        a[i] = a[n - i - 1];

        a[n - i - 1] = temp;

    }

}

### 冒泡排序

使用冒泡排序对输入的字符串排序，要求参数的传入使用一维数组指针。

void Bubble\_Sort(char str[][100], int n)

{

    char temp[100];

    int i, j;

    for (i = 0; i < n - 1; i++)

    {

        for (j = 0; j < n - i - 1; j++)

        {

            if (strcmp(str[j], str[j + 1]) > 0)

            {

                strcpy(temp, str[j]);

                strcpy(str[j], str[j + 1]);

                strcpy(str[j + 1], temp);

            }

        }

    }

}