**数组与指针**

**一．指针与数组的联系：**

指针与数组是C语言中很重要的两个概念，它们之间有着密切的关系，利用这种关系，可以增强处理数组的灵活性，加快运行速度，本文着重讨论指针与数组之间的联系及在编程中的应用。

**1．指针与数组的关系**

　　当一个指针变量被初始化成数组名时，就说该指针变量指向了数组。如：

　　char str[20], \*ptr;

　　ptr=str;

**ptr被置为数组str的第一个元素的地址，因为数组名就是该数组的首地址，也是数组第一个元素的地址。**此时可以认为指针ptr就是数组str（反之不成立），这样原来对数组的处理都可以用指针来实现。如对数组元素的访问，既可以用下标变量访问，也可以用指针访问。

**2．指向数组元素的指针**

　　若有如下定义：

　　int a[10], \*pa;

　　pa=a;

　　则p=&a[0]是将数组第1个元素的地址赋给了指针变量p。

　　实际上，C语言中数组名就是数组的首地址，所以第一个元素的地址可以用两种方法获得：p=&a[0]或p=a。

　　这两种方法在形式上相像，其区别在于：pa是指针变量，a是数组名。值得注意的是：pa是一个可以变化的指针变量，而a是一个常数。因为数组一经被说明，数组的地址也就是固定的，因此a是不能变化的，不允许使用a＋＋、＋＋a或语句a＋=10，而pa＋＋、＋＋pa、pa＋=10则是正确的。由此可见，此时指针与数组融为一体。

**3．指针与一维数组**

　　理解指针与一维数组的关系，首先要了解在编译系统中，一维数组的存储组织形式和对数组元素的访问方法。

**一维数组是一个线形表，它被存放在一片连续的内存单元中。**C语言对数组的访问是通过数组名（数组的起始地址）加上相对于起始地址的相对量（由下标变量给出），得到要访问的数组元素的单元地址，然后再对计算出的单元地址的内容进行访问。通常把数据类型所占单元的字节个数称为扩大因子。

　　实际上编译系统将数组元素的形式a[i]转换成\*(a＋i)，然后才进行运算。对于一般数组元素的形式：<数组名>[<下标表达式>]，编译程序将其转换成：\*(<数组名>＋<下标表达式>)，其中下标表达式为：下标表达式\*扩大因子。整个式子计算结果是一个内存地址，最后的结果为：\*<地址>=<地址所对应单元的地址的内容>。由此可见，C语言对数组的处理，实际上是转换成指针地址的运算。

　　数组与指针暗中结合在一起。因此，**任何能由下标完成的操作，都可以用指针来实现，**一个不带下标的数组名就是一个指向该数组的指针。

**4．指针与多维数组**

　　用指针变量可以指向一维数组，也可以指向多维数组。但在概念上和使用上，多维数组的指针比一维数组的指针要复杂一些。

　　例如，在一个三维数组中，引用元素c[i][j][k]的地址计算最终将换成：\*(\*(\*(c＋i)＋j)＋k)。了解了多维数组的存储形式和访问多维数组元素的内部转换公式后，再看当一个指针变量指向多维数组及其元素的情况。

**1）指向数组元素的指针变量**

　　若有如下说明：

　　int a[3][4];

　　int \*p;

　　p=a;

　　p是指向整型变量的指针；p=a使p指向整型二维数组a的首地址。

　　\*(\*(p＋1)＋2)表示取a[1][2]的内容；\*p表示取a[0][1]的内容，因为p是指向整型变量的指针；**p＋＋表示p的内容加1，即p中存放的地址增加一个整型量的字节数2，**从而使p指向下一个整型量a[0][1]。

**2）指向由j个整数组成的一维数组的指针变量**

　　当指针变量p不是指向整型变量，而是指向一个包含j个元素的一维数组。如果p=a[0]，则p＋＋不是指向a[0][1]，而是指向a[1]。这时p的增值以一维数组的长度为单位。

**5．指针与字符数组**

　　C语言中许多字符串操作都是由指向字符数组的指针及指针的运算来实现的。因为对于字符串来说，一般都是严格的顺序存取方式，使用指针可以打破这种存取方式，更为灵活地处理字符串。

另外由于字符串以′\0′作为结束符，而′\0′的ASCII码是0，它正好是C语言的逻辑假值，所以可以直接用它作为判断字符串结束的条件，而不需要用字符串的长度来判断。**C语言中类似的字符串处理函数都是用指针来完成，**使程序运行速度更快、效率更高，而且更易于理解。

**二．指针与数组的区别：**

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。

指针可以随时指向任意类型的内存块，它的特征是“可变”，所以我们常用指针来操作动态内存。指针远比数组灵活，但也更危险。

下面以字符串为例比较指针与数组的特性。

**1 修改内容**

       示例1中，字符数组a的容量是6个字符，其内容为hello\0。a的内容可以改变，如a[0]= ‘X’。指针p指向常量字符串“world”（位于静态存储区，内容为world\0），常量字符串的内容是不可以被修改的。从语法上看，编译器并不觉得语句p[0]= ‘X’有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

|  |
| --- |
| char a[] = “hello”;  a[0] = ‘X’;  cout << a << endl;  char \*p = “world”;     // 注意p指向常量字符串  p[0] = ‘X’;             // 编译器不能发现该错误  cout << p << endl; |

示例1 修改数组和指针的内容

**2 内容复制与比较**

    不能对数组名进行直接复制与比较。示例2中，若想把数组a的内容复制给数组b，不能用语句 b = a ，否则将产生编译错误。应该用标准库函数strcpy进行复制。同理，比较b和a的内容是否相同，不能用if(b==a) 来判断，应该用标准库函数strcmp进行比较。

    语句p = a 并不能把a的内容复制指针p，而是把a的地址赋给了p。要想复制a的内容，可以先用库函数malloc为p申请一块容量为strlen(a)+1个字符的内存，再用strcpy进行字符串复制。同理，语句if(p==a) 比较的不是内容而是地址，应该用库函数strcmp来比较。

|  |
| --- |
| // 数组…      char a[] = "hello";      char b[10];      strcpy(b, a);           // 不能用   b = a;      if(strcmp(b, a) == 0)   // 不能用  if (b == a)  … |
| // 指针…      int len = strlen(a);      char \*p = (char \*)malloc(sizeof(char)\*(len+1));      strcpy(p,a);            // 不要用 p = a;      if(strcmp(p, a) == 0)   // 不要用 if (p == a)  … |

示例2 数组和指针的内容复制与比较

**3 计算内存容量**

    用运算符sizeof可以计算出数组的容量（字节数）。示例3（a）中，sizeof(a)的值是12（注意别忘了’\0’）。指针p指向a，但是sizeof(p)的值却是4。这是因为sizeof(p)得到的是一个指针变量的字节数，相当于sizeof(char\*)，而不是p所指的内存容量。C++/C语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。

**注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。**示例3（b）中，不论数组a的容量是多少，sizeof(a)始终等于sizeof(char \*)。

|  |
| --- |
| char a[] = "hello world";      char \*p  = a;      cout<< sizeof(a) << endl;   // 12字节      cout<< sizeof(p) << endl;   // 4字节 |

示例3（a） 计算数组和指针的内存容量

|  |
| --- |
| void Func(char a[100])      {          cout<< sizeof(a) << endl;   // 4字节而不是100字节  } |

示例3（b） 数组退化为指针