# 致未来：

来自2017.10.12日的自己

建立这个文档的时候，我认为我对C语言已经有了相对整体的把握。在这个问题中，我不再零碎地罗列知识要点（第一阶段），也不仅仅是对某一部分概念进行有架构地总结（第二阶段，实际上我现在也同时处于这个阶段），更重要地，我希望在这个文档中，可以谈论更多有关于思考的问题： 可以陈述我对某一问题/概念的看法，也可以就处理某一问题展开具体的讨论；可以是朴实严谨的问题，也可以天马星空的想象；可以是对正确的验证，也可以是某个低级错误所引发的趣味之旅……

总之，我希望这个文档是以“计算机问题漫谈”的形式而存在，旨在引发我对“问题”的思考

# 关于指针数组和数组指针的讨论

**指针数组**是一个数组，数组中每一个元素都是指针。（占用一个数组的空间，有多个指针）

**数组指针**是一个数组类型的指针，本质上是一个指针，指向与其类型相同的一个数组。（只占用一个指针的空间，只有一个指针）

指针数组和数组指针写法上的 不同是由运算符的优先级所导致的。

Char \* p[10] ：【】的优先级高于\*。所以，p首先是一个数组，后规定数组元素为char指针，那么我们就定义了一个指针数组。（是数组）

而Char (\* p)[10]则改变了这种优先级，p首先与\*相结合，后规定其指向一个包含10个元素的数组。那么我们便定义了一个数组指针。

**为了方便区别，以后应该分别称指针数组和数组指针为：指针类型的数组和数组类型的指针**。

以上都是一些概念问题，虽然有些拗口，但是只要把握住符号的优先级就不难。

**下面我希望讨论的问题是：指针类型的数组和数组类型的指针分别应该在什么场合下使用？**

我们先来看看指针类型的数组应用于怎样的场合。我想首先要抓住的一点是：指针类型的数组是多个指针，而数组类型的指针是只有一个指针。那么，什么时候我们需要用到多个在内存上连续存储的指针呢？

第一个想到的应该是字符串的排序问题！

假设我现在有一个字符串数组

char str[4][LEN]=

{

“Learning C is a long term effort”,

”The beatiful things remain for learners”,

”Never give up”,

”Earn what belongs to you!”

}

现在要对其按照字母顺序进行排序，那么便可以利用到指针数组：

char \* p\_order[4]; //定义了一个指针类型的数组

for(int i=0;i<5;i++) //对p\_order[]进行初始化，每个元素指向对应字符串

p\_order[i]=&str[i];

然后，再对指针进行相应的排序（而不是对原数组）：

for()

{

for()

{

if(strcmp(**p\_order[j],p\_order[j+1])**)

………………….

}

}

最后输出p\_order[]指向的内容即可。

**思考一下，这样做有什么优点呢？为什么不直接给字符串数组排序，而要多此一举，给指向字符串的指针排序呢？？？**

这样做有两个优点：

1. 操作指针，而不操作指针指向的内容，确保了数据的安全，这一点是显而易见的。

2. 更重要的是，这样提高了程序的效率。我们只是交换指向字符串的指针，而不用交换字符串本身。这样不论字符串有多长，运行的效率都一样快。

这就好比在仓库排号，只需要给每样货物一个号码，指定他们哪个先发货哪个后发货，而不需要真的把所有货物都按照先后顺序排列好，那样就太花功夫了！！！

# 论指针

# Typedef讨论

Typedef用于简化复杂的类型标识符，提高代码的可读性，其功能上和#define有所类似，且更加灵活。

Typedef unsigned int newdata

那么newdata var就相当于unsigned int var，这很容易理解。

但是，当它与更多的标识符相结合的时候，就那么容易理解了。

比如：

例1.typedef char newdata \*

则newdata var == char \* var !

例2. typedef char newdata[4]

则newdata var == char var[4]!!

例3. typedef (double)(\* PF)(int)

则PF pfun == (double)(\*pf)(int)!!!

上面这些复杂的声明看起来比较抽象，又该如何理解呢？

秘诀在于分步骤来看。比如看例1，可以先看typedef char newdata. 那么这时候newdata就可以像char 一样使用，然后typedef char newdata \* 则容易理解为newdata == char \*。例2也是同样的道理。至于例3中的模式，比较僵硬，记住就好了！

# 算法漫谈

刚刚开始学习了几天算法，感觉又打开了新世界的大门，非常有趣，暂时也没有感觉很难。但是，这几天刷题发现了一个问题：在设计算法的时候，常常忽略了一些“特例”的存在，使得算法的思想总是保留着一些bug?

那么，有没有什么算法的设计规范，能够通用于普遍的算法呢？也就是说，在设计算法的时候，要考虑一些什么因素才能够让算法出错的可能性减小？

1. 边界值。