原文地址：http://blog.csdn.net/supermegaboy/archive/2009/11/23/4855036.aspx

此文是笔者 2005 年所作《再再论指针》的修订版，与前文相比，本文主要的不同点有如下几点：

一、引用 C/C++标准的条款去阐述原理。C 与 C++标准共有四个已发行的正式版本，分别是 C89/C90、C99、C++98 和 C++2003，为了避免重复的条款引用，在文字或语义相同的情况下，笔者只引用 C99 的条款，遇到不同的情况时再分别引用。
二、加入了 C++的内容。
三、增加或者合并了一些章节，同时修正了一些行文错误。

指针是 C/C++的灵魂！它是 C/C++众多引人入胜的特性中的一朵奇葩。与底层操作的亲密接触是指针与生俱来的优点，利用指针可以写出许多短小精悍、效率极高的代码。它是 C/C++一把无可替代的利器，凭着这把利器，C/C++与其它高级语言相比至少在效率方面胜人一筹。

但是，由于数组与指针的原理与使用方式跟人们通常的思维习惯有较大差别，需要花较多的时间进行消化，这使得对数组与指针的偏见和误解成为了普遍存在的现象，更出现了避免使用指针的思潮，笔者认为这是非常不可取的。指针是如此犀利，正是它才使得 C/C++语言威猛无比。如果就这样把它放弃了，那么 C/C++就算是白学了。我们应当让指针成为你手中那把砍掉索伦手指上魔戒的举世无双的纳西尔圣剑，而不是你心中永远的魔戒。

与众多初学者一样，笔者对数组与指针的理解也经历了漫长的过程。初学 C 的时候，笔者作为谭书的受害者之一，与其它初学者一样脑袋中充满了对数组与指针各种各样的误解。后来随着对 C/C++理 解的深入，逐渐发现谭书中存在大量的谬误与漏洞，从此开始了纠偏的历程。这是一段痛并快乐着的过程，痛是由于愤恨被谋杀了一段宝贵的时光，快乐是因为重新 找到了方向，相信不少朋友也曾有过跟笔者相似的体会。在这段时间里，笔者查阅了大量的资料，也耗费了大量的时间进行深刻的思考，还跟同事、朋友、网友进行 了大量的辩论，特别是 2005 年，几乎整整一年的时间都是在激烈的辩论中渡过的，当年这些辩论的激烈程度到现在还记忆犹新，如果当时手里有把枪的话，我想我会开枪的！经过不断的思考、辩论、印证，再思考、再辩论、再印证，数组与指针的迷雾终于逐渐清晰了起来。

本文的目的，是希望通过跟各位朋友一起讨论关于数组与指针的几个关键概念及常见问题，加深对数组与指针的理解。笔者不敢奢望能够完全解开你心中的魔结，但如果通过阅读本文，能够让你在日后的数组与指针使用过程中减少失误，笔者就心满意足了。

当你阅读本文后：

如果你有不同的意见，欢迎你在评论里留下自己的见解，笔者很乐意跟你一起讨论，共同进步。

如果你觉得我说的全都是废话，那么恭喜你，你的指针已经毕业了。

如果你有太多不明白的地方，那么我介绍你先找一些关于数组与指针的读物看看，笔者推荐你阅读一本叫《C 与指针》的书，看完后再回来继续思考你的问题。

第一章数组与指针概念剖析

数组与指针生来就是双胞胎，多数人就是从数组的学习开始指针的旅程的。在学习的过程中，很自然就会经常听到或见到关于数组与指针的各种各样的看法，下面我节选一些在各种论坛和文章里经常见到的文字：

“一维数组是一级指针”
“二维数组是二级指针”
“数组名是一个常量指针”
“数组名是一个指针常量”

这些文字看起来非常熟悉吧？类似的文字还有许多。不过非常遗憾，这些文字都是错误的，实际上数组名永远都不是指针！这个结论也许会让你震惊，但它的确是事 实。但是，在论述这个问题之前，首先需要解决两个问题：什么是指针？什么是数组？这是本章的主要内容，数组名是否指针这个问题留在第二章进行讨论。看到这 里，也许有人心里就会嘀咕了，这么简单的问题还需要说吗？int *p, a[10];不就是指针和数组吗？但是，笔者在过往的

讨论过程中，还真的发现有不少人对这两个概念远非清晰，这会妨碍对后面内容的理解，所以还是有必要先讨论一下。

什么是指针？一种普遍存在的理解是，把指针变量理解成就是指针，这种理解是片面的，指针变量只是指针的其中一种形态，但指针并不仅仅只有指针变量。一个指针，包含了两方面的涵义：实体（entity）和类型。标准是这样描述指针类型的：

6.2.5 Types

A pointer type may be derived from a function type, an object type, or an incomplete type, called the referenced type. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type T is sometimes called ‘‘pointer to T’’. The construction of a pointer type from a referenced type is called ‘‘pointer type derivation’’.

请留意第二句所说的内容：指针类型描述了这样一个对象，其值为对某种类型实体的引用。标准在这里所用的措词是指针类型描述了一个对象。

再来看看标准关于取址运算符&的规定：

6.5.3.2 Address and indirection operators

Semantics

The unary & operator returns the address of its operand. If the operand has type “type”, the result has type “pointer to type”..... Otherwise, the result is a pointer to the object or function designated by its operand.

这个条款规定，&运算符的结果是一个指针。但问题是，&表达式的结果不是对象！标准自相矛盾了吗？当然不是，这说明的是，指针的实体有对象与非对象两种形态。

我们常说的指针变量只是指针实体的对象形态，但对象与非对象两种形态合起来，才是指针的完整涵义，就是说，无论是否对象，只要是一个具有指针类型的实体，都 可以称之为指针，换言之，指针不一定是对象，也不一定是变量。后一种情况，指的是当需要产生一个指针类型的临时对象时，例如函数的传值返回或者表达式计算 产生的中间结果，由于是一个无名临时对象，因此不是变量。

在 C++中，由于引入了 OOP，增加了一种也称为“指针”的实体：类非静态成员指针，虽然也叫指针，但它却不是一般意义上的指针。C++标准是这样说的：

3.9.2 Compound types

..... Except for pointers to static members, text referring to “pointers” does not apply to pointers to members.....

接下来，该谈谈数组了。数组是一种对象，其对象类型就叫数组类型。但笔者发现有个现象很奇怪，有些人根本没有数组类型的意识，不过也的确有些书并没有将数组作为一个类型去阐述，也许原因就在于此吧。数组类型跟指针类型都属于派生类型，标准的条款：

6.2.5 Types

An array type describes a contiguously allocated nonempty set of objects with a particular member object type, called the element type. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is T, the array type is sometimes called “array of T”. The construction of an array type from an element type is called “array type derivation”.

数组类型描述了某种对象的非空集合，不允许 0 个元素，我们有时候看见某个结构定义内部定义了一个大小为 0 的数组成员，这是柔性数组成员的非标准形式，这个留在第八章讲述。数组类型的语法（注意不是数组对象的声明语法）是 element type[interger constant]，例如对于

int a[10];
a 的数组类型描述就是 int[10]。

数组名作为数组对象的标识符，是一个经过“隐式特例化”处理的特殊标识符。整数对象的标识符、浮点数的标识符等等虽然也是标识符，但数组名与之相比却有重大的区别。计算机语言存在的目的，是为了将人类的自然语言翻译为计算机能够理解的机器语言,让 人类更加容易地利用和管理各种计算机资源，易用是思想，抽象是方法，语言将计算机资源抽象成各色各样的语言符号和语言规则，数组、指针、整数、浮点数等等 这些东西本质上就是对内存操作的不同抽象。作为抽象的方法，可以归纳为两种实现，一是名字代表一段有限空间，其内容称为值；二是名字是一段有限空间的引 用，同时规定空间的长度。第一种方法被各种计算机语言普遍使用，在 C/C++中称为从左值到右值的转换。但数组不同于一般的整数、浮点数对象，它是一个聚集，无法将一个聚集看作一个值，从一个聚集中取值，在 C/C++的对象模型看来缺乏合理性，是没有意义的。在表达式计算的大多数情况中，第一种方法并不适合数组，使用第二种方法将数组名转换为某段内存空间的引用更适合。

因此，与一般标识符相比，数组名既有一般性，也有特殊性。一般性表现在其对象性质与一般标识符是一样的，这种情况下的数组名，代表数组对象，同时由于符合 C/C++的左值模型，它是一个左值，只不过是不可修改的，不可修改的原因与上一段中叙述的内容相同，通过一个名字试图修改整个聚集是没有意义的；而特殊性则反映在表达式的计算中，也就是 C/C++标准中所描述的数组与指针转换条款，在这个条款中，数组名不被转换为对象的值，而是一个符号地址。

现在来看看标准是如何规定数组与指针的转换的：

C89/90 的内容：
6.2.2.1 Lvalues and function designators

Except when it is the operand of the sizeof operator or the unary & operator, or is a character string literal used to initialize an array of character type. or is a wide string literal used to initialize an array with element type compatible with wchar-t, an lvalue that has type “array of type” is converted to an expression that has type “pointer to type” that points to the initial element of the array object and is not an lvalue.

C99 的内容：
6.3.2.1 Lvalues, arrays, and function designators

Except when it is the operand of the sizeof operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type “array of type” is converted to an expression with type “pointer to type” that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

数组类型到指针类型转换的结果，是一个指向数组首元素的类型为 pointer to type 的 指针，并且从一个左值转换成一个右值。经过转换后，数组名不再代表数组对象，而是一个代表数组首地址的符号地址，并且不是对象。特别指出的是，数组到指针 的转换规则只适用于表达式，只在这种条件下数组名才作为转换的结果代表数组的首地址，而当数组名作为数组对象定义的标识符、初始化器及作为 sizeof、&的操作数时，它才代表数组对象本身，并不是地址。

这种转换带来一个好处，对于数组内部的指针运算非常有利。我们可以用 a + 1 这种精炼的形式表示 a[1]的地址，无须用&a[1]这种丑陋的代码，实际上，&a[1]是一种代码冗余，是对代码的浪费，因为&a[1]等价于&*(a + 1)，&与*由于作用相反被抵消，实际上就是 a + 1，既然这样我们何不直接使用 a + 1 呢？撇开为了照顾人类阅读习惯而产生的可读性而言，&a[1]就是垃圾。

但是，另一方面，这种异于一般标识符左值转换的特例化大大增加了数组与指针的复杂

性，困扰初学者无数个日日夜夜的思维风暴从此拉开了帷幕！

在两个版本的转换条款中，有一点需要留意的是，两个版本关于具有数组类型的表达式有不同的描述。

C89/90 规定：
an lvalue that has type “array of type” is.....
但 C99 却规定：
an expression that has type “array of tye” is.....

C99 中去掉了 lvalue 的词藻，为什么？我们知道，数组名是一个不可修改的左值，但实际上，也存在右值数组。在 C 中，一个左值是具有对象类型或非 void 不完整类型的表达式，C 的左值表达式排除了函数和函数调用，而 C++因为增加了引用类型，因此返回引用的函数调用也属于左值表达式，就是说，非引用返回的函数调用都是右值，如果函数非引用返回中包含数组，情况会怎样？考虑下面的代码：

```
#include <stdio.h>

struct Test
{
    int a[10];
};

struct Test fun( struct Test* );

int main( void )
{
    struct Test T;
    int *p = fun( &T ).a;                       /* A */
    int (*q)[10] = &fun( &T ).a;                /* B */
    printf( "%d", sizeof( fun( &T ).a ) );      /* C*/
    return 0;
}

struct Test fun( struct Test *T )
{
    return *T;
}
```

在这个例子里，fun(&T)的返回值是一个右值，fun(&T).a 就是一个右值数组，是一个右值表达式，但 a 本身是一个左值表达式，要注意这个区别。在 C89/90 中，由于规定左值数组才能进行数组到指针的转换，因此 A 中的 fun(&T).a 不能在表达式中进行从数组类型到指针类型的转换，A 中的 fun(&T).a 是非法的，但 C99 在上述条款中不再限定左值表达式，即对这个转换不再区分左值还是右值数组，因此都是合法的；C 中的 fun(&T).a 是 sizeof 运算符的操作数，这种情况下 fun(&T).a 并不进行数组到指针的转换，因此 C 在所有 C/C++标准中都是合法的；B 初看上去仍然有点诡异，&运算符不是已经作为例外排除了数组与指针的转换吗？为什么还是非法？其实 B 违反了另一条规定，&的操作数要求是左值，而 fun(&T).a 是右值。C++继承了 C99 的观点，也允许右值数组的转换，其条款非常简单：
An lvalue or rvalue of type “array of N T” or “array of unknown bound of T” can be converted to an rvalue of type “pointer to T.” The result is a pointer to the first element of the array.

数组类型到指针类型的转换与左值到右值的转换、函数类型到指针类型的转换一起是 C/C++三条非常重要的转换规则。C++由于重载解析的需要，把这三条规则概念化了，统称为左值转换，但 C 由于无此需要，只提出了规则。符号是语言对计算机的高级抽象，但计算机并不认识符号，它只认识数值，因此一个符号要参加表达式计算必须先对其进行数值化，

三条转换规则就是为了这个目的而存在的。

看到这里，大概有些初学者已经被上述那些左值右值、对象非对象搞得稀里糊涂了。的确，数组与指针的复杂性让人望而生畏，不是一朝一夕就能完全掌握的，需要一段较长的时间慢慢消化。因此笔者才将数组与指针称为一门艺术，是的，它就是艺术！

第二章数组名是一个指针常量吗？

数组名是一个指针常量这种观点来源于数组名在表达式计算中与指针的结果等效性。例如下面的代码：

```
int a[10], *p = a, *q;
q = a + 1;
q = p + 1;
```

在效果上看，a + 1 与 p + 1 是相同的，这很容易给人一种 a 就是 p 的假象，但，这仅仅是假象。鉴于指针常量包含了指针和常量两类概念，我们可以把这个问题分开两部分进行讨论。

一、数组名是指针吗？

在《C 与指针》一书中，作者用一个著名的例子阐述了数组名与指针的不同。在一个文件中定义：int a[10];然后在另一个文件中声明：extern int *a; 笔者不在这里重复其中的原理，书中的作者试图从底层操作上阐述数组名与指针的不同点，但笔者认为这个例子存在一些不足，a 在表达式中会转换为一个非对象的符号地址，而指针 a 却 是一个对象，用一个非对象去跟一个对象比较，有“偷跑”的嫌疑，这个例子只是说明了数组名的非对象性质，只能证明对象与非对象实体在底层操作上的不同，事 实上，如上一章所述，指针也有非对象形态。笔者认为，无须从底层的角度上花费那么多唇舌，仅仅从字面上的语义就可以推翻数组名是一个指针的观点。

首先，在C / C＋＋中，数组类型跟指针类型是两种不同的派生类型，数组名跟指针是两种不同类型的实体，把数组类型的实体说成“是”另一个类型的实体，本身就是荒谬的；

其次，a + 1 在效果上之所以等同于 p + 1，是因为 a 进行了数组到指针的隐式转换，这是一个转换的过程，是 converted to 而不是 is a 的过程。如果是两个相同的事物，又怎会有转换的过程呢？当把 a 放在 a + 1 表达式中时，a 已经从一个数组名转换为一个指针，a 是作为指针而不是数组名参与运算的；

第三，a + 1 与 p + 1 是等效关系，不是等价关系。等价是相同事物的不同表现形式，而等效是不同事物的相同效果。把数组名说成是指针实际上把等效关系误解为等价关系。

因此，数组名不是指针，永远也不是，但在一定条件下，数组名可以转换为指针。

二、数组名是一个常量吗？

看见这句话有人会觉得奇怪，数组定义之后就不能改变了，数组名不就是个常量吗？在表达式中，数组名的确可以转换为一个不变的符号地址，但在C 中，不变的实体不一定是常量！而且，C/C++有常量与常量表达式之分，常量与常量表达式是两种不同的实体，但常量表达式可以作为常量使用。C/C++中的常量虽然有所不同，但都不包括数组或数组名，而且数组名也不一定是常量表达式。

请在 C90 的编译器中编译如下代码，注意不能是 C99 和 C++的，因为 C99 和 C++不再规定数组的初始化器必须是常量表达式，会看不到效果：

```
int main( void )
{
    static int a[10], b[10];
    int c[10], d[10];
    int* e[] = { a, b };        /* A */
    int* f[] = { c, d };        /* B */
    return 0;
}
```

B 为什么不能通过编译？是由于自动数组名并不是常量表达式。在 C 中，常量表达式必须是编译期的，只在运行期不变的实体不是常量表达式，请看标准的摘录：

6.6 Constant expressions

A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

c 和 d 是自动数组，首地址在编译期是不可知的，因为这样的对象在编译期还不存在；a 和 b 是静态数组，静态对象从程序开始时就已存在，因此 a 和 b 的首地址在编译期是已知的，它们都属于常量表达式中的地址常量表达式。

所以，C/C++中的数组名，都不是常量。C 中的数组名，是否常量表达式要视其存储连续性而定，全局数组、静态数组名都是常量表达式，而自动数组名不是。在 C＋＋中，由于不再规定常量表达式必须是编译期的，因此 C＋＋的数组名都是常量表达式。

第三章 数组的解剖学

C/C++的数组不同于 VB 等语言的数组，是有层次的，这个层次指的不是维度，而是象俄罗斯有名的套娃一样，一维套一维，亦即数组的嵌套，数组的元素也是数组，VB 等语言的数组与之相比更像一个平面。

数组嵌套这个现象从其它语言的角度来看有点奇特，但其实原因也很简单。C/C++的 对象模型并不视数组为某种数值的简单集合，而是对象的聚集，每个元素都是一个对象。元素为整数对象，就是整数数组，为浮点数对象，就是浮点数数组。然而， 数组本身也是一种对象，因此一个数组也能作为另一个数组的元素。当某个一维数组以一维数组作为元素时，这个一维数组每个元素都具有数组类型，这个一维数组 其实是二维数组，同理，一个以二维数组作为元素的一维数组其实是三维数组。因此，使用 C/C++数组的时候应该用数组嵌套的观点去看待。有人据此认为，C/C++的数组不是真正的数组，还有的认为 C/C++没有多维数组，这些观点都有失偏颇，与其它语言的数组相比，两者只是同一事物的不同实例，是实现方法的不同，而本质是一样的，C/C++的数组嵌套可视为对数组概念的发展。

现在来看看数组的定义：

6.5.4.2 Array declarators

Semantics

If, in the declaration “T Dl.” Dl has the form
D [constant expressionopt]

这个定义非常简单，其中 T 代表元素类型，D 代表标识符，constant expression 必须为大于 0 的常量表达式，opt 表示可选，即[]中的内容可以为空，当[]为空时叫不完整类型，表示这个数组对象的长度未知，不完整数组类型可以在程序的某个地方补充完整。细心的人马上就会发现，从形式上看，怎么只有一维数组的定义？这个形式如何定义多维数组？刚才说过，C/C++的数组是数组的嵌套，因此多维数组的定义也反映了这个本质。多维数组的定义是通过嵌套的一维数组定义构造的。对于一维数组：

T D[M]

当元素为一维数组 T[N]时，元素的类型也为数组类型，用 T[N]代替 T，则为：

T[N] D[M]

这个语法结构不符合 C/C++数组定义的语法形式，将[N]移动到[M]后，就是正式的二维数组的定义了：

T D[M][N]

其中 D[0]---D[M-1]都是一维数组，具有数组类型 T[N]。各种维度的多维数组可以用同样的嵌套方法构造出来。

一个一维数组 T[M]，经过数组到指针的转换后，类型转换为 T*，二维数组 T[M][N]转换为指针后，类型转换为 T (*)[N]，有些初学者对 T(*)[N]这种形式较难理解，怎么多了一维，形式就有这么大的差别呢，其实原理还是跟嵌套有关，二维数组为一维数组的嵌套，元素为数组类型，因此用 T[N]代替 T，则二维数组转换之后的指针类型为 T[N] *，将[N]移动到*的右边，就是 T*[N]，由于[]的优先级比*高，因此需要加括号，就成为 T(*)[N]了，否则就不是指针类型，而成了指针数组类型了。

围绕数组名，存在一些有趣的表达式，下面的内容通过讨论这些表达式中较为重要的几个，来加深对数组的理解。对于二维数组：

T a[M][N]

a： 表达式中的 a 的类型转换为 T (*)[N]，代表数组的首地址；
&a：是一个指向二维数组对象的指针，类型为 T (*)[M][N]。在 C 标准出现之前，一些早期的实现并不允许&a，因为这些编译器认为此时的 a 转换为一个右值，而&运算符要求一个左值，因此非法。C 标准委员会鉴于对象的概念已经得到了扩展，而且允许&a 并没有害处，因此把&运算符作为一个例外写进了数组到指针的转换条款中。这种情况下的 a 代表数组对象，&a 表示对数组对象取地址，因此&a 的结果跟 a 是相同的，但类型不同。

笔者曾经见过某些观点认为，&a 才是数组的首地址，不是 a。这个观点初看起来似乎很有道理，一个数组对象的引用，不正是首地址吗？但实际上这种论述是不符合标准的，数组到指针的转换条款规定，当产生一个 points to the initial element of the array object 的时候，其前提是由 array of type 到 pointer to type 的转换，但&a 的类型属于 pointer to array of type，不是 pointer to type，因此真正代表数组首地址的是 a 本身，不是&a。

&a[0][0]：这是数组首元素的地址。&a[0][0]常常被误解为数组 a 的首地址，其实 a[0][0]只不过由于位置特殊，其地址值才与 a 相同，&a[0][0]是一个 T 类型对象的引用，不是一个数组对象的引用，而且其类型不是由 array of type 转换得来的，因此其意义不是数组首地址。
a[i]（其中 i >= 0 && i < M）：从数组嵌套的观点来看，a 是一个一维数组，元素的类型为数组类型，因此 a[i]的类型为 T[N]，在表达式中转换为 T*，是第 i 个一维数组的首地址。
a + 1：a 隐式转换为指针类型 T(*)[N]然后加 1，请记住指针加法是以指针指向对象的大小为步长的，因此 a + 1 将跨过 N * sizeof(T)个字节。
&a + 1：与 a + 1 同理，&a 类型为 T(*)[M][N]，因此&a + 1 的步长为 M * N * sizeof(T)。

第四章 []运算符的本质

下标运算符[]一直被作为数组的专有运算符来介绍，经过长年的应用，人们也早已对这个用法习以为常，视为跟每天的午餐一样稀松平常的事情。当你很遐意地写下 a[0]表达式的时候，如果抽空回过头来看看标准中关于下标运算符的条款，你很可能会大吃一惊：

6.5.2.1 Array subscripting

Constraints
One of the expressions shall have type ‘‘pointer to object type’’, the other expression shall have integer type, and the result has type ‘‘type’’.

其中一个表达式具有指针类型，而不是数组类型！为什么会这样呢？如果规定为数组类型，由于表达式中的数组类型会隐式转换为指针类型，两个条款就会产生矛盾， 当然，可以将下标运算符也作为转换规则的例外，但直接规定为指针类型显然能带来更多的好处，而且，既然数组类型能够转换为指针类型，却不让指针使用下标运 算符，会显得无可理喻。从条款的角度来讲，下标运算符其实是指针运算符。

另一个表达式的类型是 integer，这意味着表达式的值可以是负数，这是由于指针运算里包含了减法的缘故，但是要注意不应该发生越界的行为。

在条款的上下文中，并没有规定[]运算符两个操作数的顺序，这意味着即使调换两个操作数的位置，也没有违反标准。这现象还可以从另一个角度进行分析，在表达式中，D[N]会转换为等价表达式*(D + N)，把 D 和 N 的位置调换，就成了*(N + D)，就是 N[D]了。

考虑如下代码：

```
int a[10]，*p = a;
p[0] = 10;
( p + 1 )[0] = 20;
0[p + 1] = 10;
( &a )[0][0] = 20;
0[&a][0] = 30;
0[0[&a]] = 40;
a[0] = “0123456789ABCDEF”[0];
```

下面对各个表达式进行解释：

p[0]：就是 a[0]；

(p + 1)[0]：p 移动一个 int 的距离，就是 a[1]；

0[p + 1]：就是(p + 1)[0]；

(&a)[0][0]：这个表达式有点古怪，a 的类型是 int[10]，&a 就是 int(*)[10]，是一个指向具有 10 个 int 元素的一维数组的指针，(&a)[0]就是&a 指向的第 0 个元素，类型为 int[10]，因此(&a)[0][0]就是(&a)[0]的第 0 个元素。

0[&a][0]：把第一维的 0 与&a 调换一下，就是 0[&a][0]；

0[0[&a]]：再调换 0[&a]与第二维[0]中的 0，就成了 0[0[&a]]，跟(&a)[0][0]等价。

最后一个表达式”0123456789ABCDEF”[0]是一个常用的技巧，它可以快速将一个数字转换为 16 进制字符。”0123456789ABCDEF”是一个字符串字面量，类型是 char[17]（在 C 中）或者 const char[17]（在 C++中），转换后的指针类型分别为 char*和 const char*，因此”0123456789ABCDEF”[0]就是第 0 个元素’0’。这个技巧常常用在进制转换中，以下代码将一个长整数的内存映像转换为 16 进制表示：

```
char* convert( unsigned long Value )
{
        static char Buffer[sizeof( unsigned long ) * 2 + 1];
        int i;
        for( i = sizeof( unsigned long ) * 2 - 1; i >= 0; --i )
        {
                Buffer[i] = "0123456789ABCDEF"[Value % 16];
                Value /= 16;
        }
        return Buffer;
}
```

当然，笔者在这里介绍这些古怪的表达式仅仅为了对下标运算符进行一些探讨，并非鼓励人们编写这样的代码。但在某些情况下，形如"0123456789ABCDEF"[Value%16]这样的表达式仍然是一个很好的选择，与下面的代码相比：

```
Remainder = Value % 16;
if( Remainder >= 10 ) Buffer[i] = 'A' + Remainder - 10;
else Buffer[i] = '0' + Remainder;
```

前者显然更加简明、精练，更容易阅读，所以，应根据不同的情况进行取舍。代码中使

用了除法和求余运算，有些人很喜欢把这些运算直接用移位代替，以追求极速。 但现代编译器对代码的优化已经非常出色，乘除运算与直接移位之间的效率差别已经小到几乎可以忽略不计的程度，除非在需要进行大量数学运算或对效率极其敏感 的场合，否则所提高的那么一点微末的速度是无法弥补可读性的损失的。在可读性、空间及效率之间应进行均衡的选择，而不是盲目追求极端。

第五章 字符串字面量---一个特殊的数组

字符串字面量（string literal）是一段双引号括起来的多字节字符序列，C/C++将其实现为具有静态存储连续性的字符数组。初学者（包括不少书籍）常将其称为字符串常量，但这说法只在 C++成立，C 中不成立。C 中的常量只包括下列四种：

6.4.4 Constants

Syntax
constant:
integer-constant
floating-constant
enumeration-constant
character-constant

分别是整数常量、浮点常量、枚举常量和字符常量，并不包括字符串字面量。但由于字符串字面量具有静态存储连续性数组类型，并且在表达式中它会根据数组到指针的隐式转换规则转换为一个代表数组首地址的右值指针，因此 C 中的字符串字面量的首地址及各元素的地址都是地址常量表达式，但字符串字面量本身不是常量，也不是常量表达式。

而 C++的情形有所不同，C++将字符串字面量归入了常量当中：

2.13 Literals

There are several kinds of literals.21)

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal

21) The term “literal” generally designates, in this International Standard, those tokens that are called “constants” in ISO C.

因此 C++中的字符串字面量才可称为字符串常量，而且首地址及各元素地址跟 C 一样，都是地址常量表达式。

字符串字面量在 C 中具有数组类型 char[N]，在 C++中则为 const char[N]，在表达式中当发生数组到指针的转换时，对应的等效指针类型分别是 char*和 const char*，因此，在 C 中，char *p = “ABCDEF”是合法的，但让人惊奇的是，上述语句在 C++中也是合法的！看起来一个 pointer to const char 指针被赋予了 pointer to char 指针，似乎违反了 C++中指针转换的 more cv-qualified 原则。其实字符串字面量在 C++中存在两种转换，一种转换依据当前上下文环境，另一种遵循数组到指针的转换，C++标准的内容：

2.13.4 String literals

……..An ordinary string literal has type “array of n const char” and static storage duration (3.7), where n is the size of the string as defined below, and is initialized with the given characters.

4.2 Array-to-pointer conversion

A string literal (2.13.4) that is not a wide string literal can be converted to an rvalue of type “pointer to char”; a wide string literal can be converted to an rvalue of type “pointer to wchar_t”. In either case, the result is a pointer to the first element of the array. This conversion is considered only when there is an explicit appropriate pointer target type, and not when there is a general need to convert from an lvalue to an rvalue. [Note: this conversion is deprecated. See Annex D.] For the purpose of ranking in overload resolution (13.3.3.1.1), this conversion is considered an array-to-pointer conversion followed by a qualification conversion (4.4). [Example: "abc" is converted to “pointer to const char” as an array-to-pointer conversion, and then to “pointer to char” as a qualification conversion.]

在具有显而易见的合适指针目标类型的情况下，例如上述 char *p = “ABCDEF”，字符串字面量被转换为 char*而不是 const char*类型的指针，这个转换实际上是对旧有代码的兼容，是一个特例，而且被指定为 deprecated 的，将在未来的版本中予以废弃，有些编译器会产生一条提示这是废弃转换的警告。而在函数重载解析中，字符串字面量遵循数组到指针的转换，同时后跟一个限定修饰的转换。

虽然字符串字面量在 C 中类型为 char[N]，在 C++中类型为 const char[N]，但并不说明 C 中的字符串字面量可以修改，C++的不可以。字符串字面量是否可以修改与实现数组的类型无关，C 之所以没有规定为 const char[N]，还是出于对旧代码的兼容，而 C++规定为 const char[N]的原因之一是比 C 更严格的类型安全。无论 C 与 C++都规定对字符串字面量的修改是未定义的，编译器可以自行处理，也的确存在一些允许修改字符串字面量的编译器，例如老一代的编译器 TC，编译器不管是否允许修改字符串字面量，都没有违反标准。

对于那些允许修改字符串字面量的编译器，必须考虑这样一个问题，当代码在不同的上下文中引用了同一个字符串字面量时，如果其中一处修改了该字面量，就会影响 其它地方的引用。解决方法是允许同一个字面量的多个实例，这样不同上下文之间不会互相干扰，标准把这个问题的决定权留给了编译器：

6.4.5 String literals

It is unspecified whether these arrays are distinct provided their elements have the appropriate values.

在 C 中，由于字符串字面量不是常量，而且 const 限定的变量不是常量表达式（C 中的常量表达式必须是编译期的），因此所有的常量和常量表达式都是右值。但 C++将字符串字面量归入常量，将 const 限定的变量归入常量表达式，这意味着在 C++中存在左值常量和左值常量表达式。

C 与 C++在这方面的差异反映出两者对待常量的不同视角。C 认为常量是不应该拥有存储空间的，这是非常传统的观点；而 C++把常量的概念延伸到了对象模型，是对对象概念的有益扩展，但同时也带来了一些问题，一个具有对象性质的实体，难以避免存在某些合法或不合法的手段去修改其内容，这种行为常常令常量对象的常量性质处于尴尬的境地，由此也催生了常量折叠这一类巧妙的折中。

第六章 指针与 const

const 一词在字面上来源于常量 constant，const 对象在 C/C++中是有不同解析的，如第二章所述，在 C 中常量表达式必须是编译期，运行期的不是常量表达式，因此 C 中的 const 不是常量表达式；但在 C++中，由于去掉了编译期的限定，因此是常量表达式。

对于一个指向 const 对象的指针 pointer to const T，由于把 const 视作常量表达式，常常存在如下两种观点：

1。这是一个指向常量的指针，简称常量指针；
2。这个指针指向的内容不可改变。

这是比较粗糙的理解。虽然这个指针的类型是 pointer to const T，但不代表它指向的对象真的是一个常量或者不可改变，例如：

```
int i = 10;
const int *p = &i;
i = 20;
```

p 指向的对象 i 明显不是常量，虽然 p 指向 i，但 i 的值依然可以改变。对于这个现象，C++标准有明确的论述：

7.1.5.1 The cv-qualifiers

a pointer or reference to a cv-qualified type need not actually point or refer to a cv-qualified object, but it is treated as if it does;

其中 cv 指的是 const 和 volatile，const 和 volatile 叫 type qualifier，类型限定词。const T 只是类型假定，并非指出该对象是什么，这个对象也许是 const 限定的，也许不是。既然上述两种看法都是不恰当的，pointer to const T 又应如何看待呢？一种比较好的理解是，将其视作一条访问路径。对一个对象进行取值或者修改操作，可以有很多种方法，每种方法都相当于一条能够对对象进行访问的路径，例如：

```
int i = 10, k;
const int *p = &i;
int *q = &i;
i = 20;
*q = 30;
k = *p;
```

通过*q、*p 和标识符 i 都能访问 i 所代表的整数对象，它们可以视作三条路径，i 和*q 能够修改该整数对象的值，这两条路径是可写可读的；但*p 不能写，因为 p 指向的对象被假定为 const，从 p 的角度看来，*p 是只读的，不能通过 p 修改它指向的对象。因此，一个 pointer to const T 指针的确切意义，不是指向常量或者指向的对象不可改变，而是指不能通过这个指针去修改其指向的对象，无论这个对象是否 const，它只指出一条到该对象的只读路径，但存在其它路径可以修改该对象。这种理解，在标准中是有根据的：

7.1.5.1 The cv-qualifiers

a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path.

上述条款对访问路径进行了一个清晰的描述。

一个 pointer to T 类型的指针，可以赋值给一个 pointer to const T 类型的指针，这是众所周知的语法规则。笔者曾经一度认为，两者之所以可以赋值，是基于指针的相容性原理，以为两者是相容的，后来翻阅了 C/C++的标准，才认识到这种解释其实是错误的，从相容性原理来说，两者恰恰是不相容的。C 标准关于指针的相容性是这样规定的：

6.7.5.1 Pointer declarators

For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.

两个相容的指针，既要有同一的限定修饰词，所指向的类型也要相容的。而两个相容的类型要符合如下规定：

6.2.7 Compatible type and composite type

Two types have compatible type if their types are the same.

两个相同的类型才具有相容性，那么 cont T 和 T 是否两种相同的类型呢？再看如下条款：

6.2.5 Types

The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.

一个类型的限定和非限定版本是同一种类类型的具有同一表示范围及对齐需求的不同类型。这就是说，const T 和 T 不是相同的类型，两者不相容，于是，虽然 pointer to const T 与 pointer to T 具有同一的限定修饰（都没有限定词），但所指向的对象类型不是相容的类型，因此 pointer to const T 与 pointer to T 是不相容的指针类型。

既然两者不相容，又是什么原因导致它们可以赋值呢？再查阅 C 标准关于赋值运算符的规定，发现有这么个条款：

6.5.16.1 Simple assignment

Constraints

One of the following shall hold:
……
— both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

噢，其实原因在这里！左操作数所指向的类型要包含右操作数所指向类型的所有限定词。pointer to const T 比 pointer to T 多一个 const，因此可以将 pointer to T 赋值给 pointer to const T，但反过来不行。通俗一点说，就是左操作数要比右操作数更严格。C++中的规定与 C 有点不同，C++标准去掉了这一条款，代之以 more cv-qualified 的概念，一个 pointer to cv1 T 的指针，要转换为一个 pointer to cv2 T 的指针，条件是 cv2 比 cv1 要更 cv 限定化。

要注意的一点是，这条赋值运算符的规则只适用于 pointer to qualified or unqualified type，不能延伸到 pointer to pointer to qualified or unqualified type 及更高级别的指针类型，例如：

```
int i = 10;
const int *p = &i;            /* A */
int *q = &i;
const int **p1 = &q;          /* B */
```

A 合法，但 B 不合法。虽然 p1 与&q 都是 unqualified 的，但 p1 指向的对象类型为 pointer to const int，&q 指向的类型为 pointer to int，如前所述，两者是不相容类型，不符合两操作数必须指向相容类型的规定，因此赋值非法。

根据上述规则，一个 pointer to const T 不能赋予 pointer to T，但是，一个 const pointer 却能赋予 non-const pointer，例如：

```
int i;
```

```
int * const p = &i;
int *q;
q = p;                  /* A */
```

A 合法，这种情况并不属于赋值运算符的规则之内，它遵循的是另一个条款：左值转换。一个被限定修饰的左值，在进行左值转换之后，右值具有左值的非限定修饰类型：

6.3.2 Other operands
6.3.2.1 Lvalues, arrays, and function designators

Except when it is the operand of the sizeof operator, the unary & operator, the ++ operator, the -- operator, or the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue). If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; otherwise, the value has the type of the lvalue.

p 的值具有 p 的非限定修饰类型 int*，与 q 类型相容，因此赋值合法。对于 C++，基本上与 C 相同，但有一个例外，就是右值类对象，由于右值类对象仍然是一个对象，C++规定右值类对象具有与左值相同的限定修饰词。

指针与 const 的结合能够产生一些比较复杂的声明，例如：

```
const int * const *** const ** const p;
```

这是一个较为复杂的指针声明符与 const 限定修饰词的组合，声明符部分嵌套了六次，中间还带有两个 const，如何辨认哪一级是 const，哪一级不是呢？一旦明白了其中的原理，其实是非常简单的。第一和最后一个 const 大家都已经很熟悉的了。对于藏在一堆*号中的 const，有一个非常简单的原则：const 与左边最后一个声明说明符之间有多少个*号，那么就是多少级指针是 const 的。例如从右数起第二个 const，它与 int 之间有 4 个*号，那么 p 的四级部分就是 const 的，下面的赋值表达式是非法的：

```
**p = (int *const***)10;
```

但下面的赋值是允许的：

```
***p=(int*const**)10;
```

从左边数起第二个 const，它与 int 之间有 1 个*，那么 p 的一级部分是 const 的，也就是*****p = (int*const***const*)10;是非法的。

第七章 右左法则----复杂指针解析

首先看看如下一个声明：

```
int* ( *( *fun )( int* ) )[10];
```

这是一个会让初学者感到头晕目眩、感到恐惧的函数指针声明。在熟练掌握 C/C++的声明语法之前，不学习一定的规则，想理解好这类复杂声明是比较困难的。

C/C++所有复杂的声明结构，都是由各种声明嵌套构成的。如何解读复杂指针声明？右左法则是一个很著名、很有效的方法。不过，右左法则其实并不是 C/C++标准里面的内容，它是从 C/C++标准的声明规定中归纳出来的方法。C/C++标准的声明规则，是用来解决如何创建声明的，而右左法则是用来解决如何辩识一个声明的，从嵌套的角度看，两者可以说是一个相反的过程。右左法则的英文原文是这样说的：

The right-left rule: Start reading the declaration from the innermost parentheses, go right, and then

go left. When you encounter parentheses, the direction should be reversed. Once everything in the parentheses has been parsed, jump out of it. Continue till the whole declaration has been parsed.

这段英文的翻译如下：

右左法则：首先从最里面的圆括号看起，然后往右看，再往左看。每当遇到圆括号时，就应该掉转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程直到整个声明解析完毕。

笔者要对这个法则进行一个小小的修正，应该是从未定义的标识符开始阅读，而不是从括号读起，之所以是未定义的标识符，是因为一个声明里面可能有多个标识符，但未定义的标识符只会有一个。

现在通过一些例子来讨论右左法则的应用，先从最简单的开始，逐步加深：

int (*func)(int *p);

首先找到那个未定义的标识符，就是 func，它的外面有一对圆括号，而且左边是一个*号，这说明 func 是一个指针，然后跳出这个圆括号，先看右边，也是一个圆括号，这说明(*func)是一个函数，而 func 是一个指向这类函数的指针，就是一个函数指针，这类函数具有 int*类型的形参，返回值类型是 int。

int (*func)(int *p, int (*f)(int*));

func 被一对括号包含，且左边有一个*号，说明 func 是一个指针，跳出括号，右边也有个括号，那么 func 是一个指向函数的指针，这类函数具有 int *和 int (*)(int*)这样的形参，返回值为 int 类型。再来看一看 func 的形参 int (*f)(int*)，类似前面的解释，f 也是一个函数指针，指向的函数具有 int*类型的形参，返回值为 int。

int (*func[5])(int *p);

func 右边是一个[]运算符，说明 func 是一个具有 5 个元素的数组，func 的左边有一个*，说明 func 的元素是指针，要注意这里的*不是修饰 func 的，而是修饰 func[5]的，原因是[]运算符优先级比*高，func 先跟[]结合，因此*修饰的是 func[5]。跳出这个括号，看右边，也是一对圆括号，说明 func 数组的元素是函数类型的指针，它所指向的函数具有 int*类型的形参，返回值类型为 int。

int (*(*func)[5])(int *p);

func 被一个圆括号包含，左边又有一个*，那么 func 是一个指针，跳出括号，右边是一个[]运算符号，说明 func 是一个指向数组的指针，现在往左看，左边有一个*号，说明这个数组的元素是指针，再跳出括号，右边又有一个括号，说明这个数组的元素是指向函数的指针。总结一下，就是：func 是一个指向数组的指针，这个数组的元素是函数指针，这些指针指向具有 int*形参，返回值为 int 类型的函数。

int (*(*func)(int *p))[5];

func 是一个函数指针，这类函数具有 int*类型的形参，返回值是指向数组的指针，所指向的数组的元素是具有 5 个 int 元素的数组。

要注意有些复杂指针声明是非法的，例如：

int func(void) [5];

func 是一个返回值为具有 5 个 int 元素的数组的函数。但 C 语言的函数返回值不能为数组，这是因为如果允许函数返回值为数组，那么接收这个数组的内容的东西，也必须是一个数组，但 C/C++语言的数组名是一个不可修改的左值，它不能直接被另一个数组的内容修改，因此函数返回值不能为数组。

int func[5](void);

func 是一个具有 5 个元素的数组，这个数组的元素都是函数。这也是非法的，因为数组

的元素必须是对象，但函数不是对象，不能作为数组的元素。

实际编程当中，需要声明一个复杂指针时，如果把整个声明写成上面所示这些形式，将对可读性带来一定的损害，应该用 typedef 来对声明逐层分解，增强可读性。

typedef 是一种声明，但它声明的不是变量，也没有创建新类型，而是某种类型的别名。typedef 有很大的用途，对一个复杂声明进行分解以增强可读性是其作用之一。例如对于声明：

```
int (*(*func)(int *p))[5];
```

可以这样分解：

```
typedef    int (*PARA)[5];
typedef PARA (*func)(int *);
```

这样就容易看得多了。

typedef 的另一个作用，是作为基于对象编程的高层抽象手段。在 ADT 中，它可以用来在 C/C++和现实世界的物件间建立关联，将这些物件抽象成 C/C++的类型系统。在设计 ADT 的时候，我们常常声明某个指针的别名，例如：

```
typedef struct node * list;
```

从 ADT 的角度看，这个声明是再自然不过的事情，可以用 list 来定义一个列表。但从 C/C++语法的角度来看，它其实是不符合 C/C++声明语法的逻辑的，它暴力地将指针声明符从指针声明器中分离出来，这会造成一些异于人们阅读习惯的现象，考虑下面代码：

```
const struct node *p1;
typedef struct node *list;
const list p2;
```

p1 类型是 const struct node*，那么 p2 呢？如果你以为就是把 list 简单“代入”p2，然后得出 p2 类型也是 const struct node*的结果，就大错特错了。p2 的类型其实是 struct node * const p2，那个 const 限定的是 p2，不是 node。造成这一奇异现象的原因是指针声明器被分割，标准中规定：

6.7.5.1 Pointer declarators

Semantics

If in the declaration ‘‘T D1’’, D1 has the form * type-qualifier-listopt D and the type specified for ident in the declaration ‘‘T D’’ is ‘‘derived-declarator-type-list T’’then the type specified for ident is ‘‘derived-declarator-type-list type-qualifier-list pointer to T’’ For each type qualifier in the list, ident is a so-qualified pointer.

指针的声明器由指针声明符*、可选的类型限定词 type-qualifier-listopt 和标识符 D 组成，这三者在逻辑上是一个整体，构成一个完整的指针声明器。这也是多个变量同列定义时指针声明符必须紧跟标识符的原因，例如：

```
int *p, q, *k;
```

p 和 k 都是指针，但 q 不是，这是因为*p、*k 是一个整体指针声明器，以表示声明的是一个指针。编译器会把指针声明符左边的类型包括其限定词作为指针指向的实体的类型，右边的限定词限定被声明的标识符。但现在 typedef struct node *list 硬生生把*从整个指针声明器中分离出来，编译器找不到*，会认为 const list p2 中的 const 是限定 p2 的，正因如此，p2 的类型是 node * const 而不是 const node*。

虽然 typedef struct node* list 不符合声明语法的逻辑，但基于 typedef 在 ADT 中的重要作用以及信息隐藏的要求，我们应该让用户这样使用 list A，而不是 list *A，因此在 ADT 的设计中仍应使用上述 typedef 语法，但需要注意其带来的不利影响。

第八章 柔性数组成员

在讲述柔性数组成员之前，首先要介绍一下不完整类型（incomplete type）。不完整类型是这样一种类型，它缺乏足够的信息例如长度去描述一个完整的对象。

6.2.5 Types

incomplete types (types that describe objects but lack information needed to determine their sizes).

C 与 C++关于不完整类型的语义是一样的。

基本上没有什么书介绍过不完整类型，很多人初次遇到这个概念时脑袋会一片空白。事实上我们在实际的工程设计中经常使用不完整类型，只不过不知道有这么个概念而已。前向声明就是一种常用的不完整类型：

```
class base;
struct test;
```

base 和 test 只给出了声明，没有给出定义。不完整类型必须通过某种方式补充完整，才能使用它们进行实例化，否则只能用于定义指针或引用，因为此时实例化的是指针或引用本身，不是 base 或 test 对象。

一个未知长度的数组也属于不完整类型：

```
extern int a[];
```

extern 不能去掉，因为数组的长度未知，不能作为定义出现。不完整类型的数组可以通过几种方式补充完整才能使用，大括号形式的初始化就是其中一种方式：

```
int a[] = { 10, 20 };
```

柔性数组成员（flexible array member）也叫伸缩性数组成员，它的出现反映了 C 程序员对精炼代码的极致追求。这种代码结构产生于对动态结构体的需求。在日常的编程中，有时候需要在结构体中存放一个长度动态的字符串，一般的做法，是在结构体中定义一个指针成员，这个指针成员指向该字符串所在的动态内存空间，例如：

```
struct test
{
        int a;
        double b;
        char *p;
};
```

p 指向字符串。这种方法造成字符串与结构体是分离的，不利于操作，如果把字符串跟结构体直接连在一起，不是更好吗？于是，可以把代码修改为这样：

```
char a[] = “hello world”;
struct test *PntTest = ( struct test* )malloc( sizeof( struct test ) + strlen( a ) + 1 );
strcpy( PntTest + 1, a );
```

这样一来，(char*)(PntTest + 1)就是字符串“hello world”的地址了。这时候 p 成了多余的东西，可以去掉。但是，又产生了另外一个问题：老是使用(char*)(PntTest + 1)不方便。如果能够找出一种方法，既能直接引用该字符串，又不占用结构体的空间，就完美了，符合这种条件的代码结构应该是一个非对象的符号地址，在结构体的尾部放置一个 0 长度的数组是一个绝妙的解决方案。不过，C/C++标准规定不能定义长度为 0 的数组，因此，有些编译器就把 0 长度的数组成员作为自己的非标准扩展，例如：

```
struct test
{
        int a;
        double b;
        char c[0];
};
```

c 就叫柔性数组成员，如果把 PntTest 指向的动态分配内存看作一个整体，c 就是一个长度可以动态变化的结构体成员，柔性一词来源于此。c 的长度为 0，因此它不占用 test 的空间，同时 PntTest->c 就是“hello world”的首地址，不需要再使用(char*)(PntTest + 1)这么丑陋的语法了。

鉴于这种代码结构所产生的重要作用，C99 甚至把它收入了标准中：

6.7.2.1 Structure and union specifiers

As a special case, the last element of a structure with more than one named member may have an incomplete array type; this is called a flexible array member.

C99 使用不完整类型实现柔性数组成员，标准形式是这样的：

```
struct test
{
        int a;
        double b;
        char c[];
};
```

c 同样不占用 test 的空间，只作为一个符号地址存在，而且必须是结构体的最后一个成员。柔性数组成员不仅可以用于字符数组，还可以是元素为其它类型的数组，例如：

```
struct test
{
        int a;
        double b;
        float c[];
};
```

应当尽量使用标准形式，在非 C99 的场合，可以使用指针方法。有些人使用 char a[1]，这是非常不可取的，把这样的 a 用作柔性数组成员会发生越界行为，虽然 C/C++标准并没有规定编译器应当检查越界，但也没有规定不能检查越界，为了一个小小的指针空间而牺牲移植性，是不值得的。

第九章 C99 可变长数组 VLA 详解

C90 及 C++的数组对象定义是静态联编的，在编译期就必须给定对象的完整信息。但在程序设计过程中，我们常常遇到需要根据上下文环境来定义数组的情况，在运行期才能确知数组的长度。对于这种情况，C90 及 C++没有什么很好的办法去解决（STL 的方法除外），只能在堆中创建一个内存映像与需求数组一样的替代品，这种替代品不具有数组类型，这是一个遗憾。C99 的可变长数组为这个问题提供了一个部分解决方案。

可变长数组（variable length array，简称 VLA）中的可变长指的是编译期可变，数组定义时其长度可为整数类型的表达式，不再象 C90/C++那样必须是整数常量表达式。在 C99 中可如下定义数组：

```
int n = 10, m = 20;
char a[n];
int b[m][n];
```

a 的类型为 char[n]，等效指针类型是 char*，b 的类型为 int[m][n]，等效指针类型是 int(*)[n]。int(*)[n]是一个指向 VLA 的指针，是由 int[n]派生而来的指针类型。

由此，C99 引入了一个新概念：可变改类型（variably modified type，简称 VM）。一个含有源自 VLA 派生的完整声明器被称为可变改的。VM 包含了 VLA 和指向 VLA 的指针，注意 VM 类型并没有创建新的类型种类，VLA 和指向 VLA 的指针仍然属于数组类型和指针类型，是数组类型和指针类型的扩展。

一个 VM 实体的声明或定义，必须符合如下三个条件：

1。代表该对象的标识符属于普通标识符（ordinary identifier）；
2。具有代码块作用域或函数原型作用域；
3。无链接性。

Ordinary identifier 指的是除下列三种情况之外的标识符：

1。标签（label）；
2。结构、联合和枚举标记（struct tag、uion tag、enum tag）；
3。结构、联合成员标识符。

这意味着 VM 类型的实体不能作为结构、联合的成员。第二个条件限制了 VM 不能具有文件作用域，存储连续性只能为 auto，这是因为编译器通常把全局对象存放于数据段，对象的完整信息必须在编译期内确定。

VLA 不能具有静态存储周期，但指向 VLA 的指针可以。

两个 VLA 数组的相容性，除了满足要具有相容的元素类型外，决定两个数组大小的表达式的值也要相等，否则行为是未定义的。

下面举些实例来对数种 VM 类型的合法性进行说明：

```
#include <stdio.h>

int n = 10;
int a[n];              /*非法，VM 类型不能具有文件作用域*/
int (*p)[n];           /*非法，VM 类型不能具有文件作用域*/
struct test
{
        int k;
        int a[n];        /*非法，a 不是普通标识符*/
        int (*p)[n];     /*非法，p 不是普通标识符*/
};

int main( void )
{
        int m = 20;
        struct test1
        {
                int k;
                int a[n];              /*非法，a 不是普通标识符*/
                int (*p)[n];           /*非法，a 不是普通标识符*/
        };
        extern int a[n];           /*非法，VLA 不能具有链接性*/
        static int b[n];           /*非法，VLA 不能具有静态存储周期*/
        int c[n];                  /*合法，自动 VLA*/
        int d[m][n];               /*合法，自动 VLA*/
        static int (*p1)[n] = d;   /*合法，静态 VM 指针*/
        n = 20;
        static int (*p2)[n] = d;   /*未定义行为*/
        return 0;
}
```

一个 VLA 对象的大小在其生存期内不可改变，即使决定其大小的表达式的值在对象定义之后发生了改变。有些人看见可变长几个字就联想到 VLA 数组在生存期内可自由改变大小，这是误解。VLA 只是编译期可变，一旦定义就不能改变，不是运行期可变，运行期可变的数组叫动态数组，动态数组在理论上是可以实现的，但付出的代价可能太大，得不偿失。考虑如下代码：

```
#include <stdio.h>

int main( void )
{
        int n = 10, m = 20;
        char a[m][n];
        char (*p)[n] = a;
        printf( “%u %u”, sizeof( a ), sizeof( *p ) );
        n = 20;
        m = 30;
        printf( “\n” );
        printf( “%u %u”, sizeof( a ), sizeof( *p ) );
        return 0;
}
```

虽然 n 和 m 的值在随后的代码中被改变，但 a 和 p 所指向对象的大小不会发生变化。

上述代码使用了运算符 sizeof，在 C90/C++中，sizeof 从操作数的类型去推演结果，不对操作数进行实际的计算，运算符的结果为整数常量。当 sizeof 的操作数是 VLA 时，情形就不同了。sizeof 必须对 VLA 进行计算才能得出 VLA 的大小，运算结果为整数，不是整数常量。

VM 除了可以作为自动对象外，还可以作为函数的形参。作为形参的 VLA，与非 VLA 数组一样，会调整为与之等效的指针，例如：

void func(int a[m][n]); 等效于 void func(int (*a)[n]);

在函数原型声明中，VLA 形参可以使用*标记，*用于[]中，表示此处声明的是一个 VLA 对象。如果函数原型声明中的 VLA 使用的是长度表达式，该表达式会被忽略，就像使用了*标记一样，下面几个函数原型声明是一样的：

```
void func( int a[m][n] );
void func( int a[*][n] );
void func( int a[ ][n] );
void func( int a[*][*] );
void func( int a[ ][*] );
void func( int (*a)[*] );
```

*标记只能用在函数原型声明中。再举个例：

```
 #include<stdio.h>

void func( int, int, int a[*][*] );

int main(void)
{
        int m = 10, n = 20;
        int a[m][n];
        int b[m][m*n];
        func( m, n, a );        /*未定义行为*/
        func( m, n, b );
    return 0;
}

void func( int m, int n, int a[m][m*n] )
{
```

```
        printf( "%u\n", sizeof( *a ) );
}
```

除了*标记外，形参中的数组还可以使用类型限定词 const、volatile、restrict 和 static 关键字。由于形参中的 VLA 被自动调整为等效的指针，因此这些类型限定词实际上限定的是一个指针，例如：

```
void func( int, int, int a[const][*] );
```

等效于

```
void func( int, int, int ( *const a )[*] );
```

它指出 a 是一个 const 对象，不能在 func 内部直接通过 a 修改其代表的对象。例如：

```
void func( int, int, int a[const][*] );

void func( int m, int n, int a[const m][n] )
{
        int b[m][n];
        a = b;              /*错误，不能通过 a 修改其代表的对象*/
}
```

static 表示传入的实参的值至少要跟其所修饰的长度表达式的值一样大。例如：

```
void func( int, int, int a[const static 20][*] );
……
int m = 20, n = 10;
int a[m][n];
int b[n][m];
func( m, n, a );
func( m, n, b );        /*错误，b 的第一维长度小于 static 20*/
```

类型限定词和 static 关键字只能用于具有数组类型的函数形参的第一维中。这里的用词是数组类型，意味着它们不仅能用于 VLA，也能用于一般数组形参。

总的来说，VLA 虽然定义时长度可变，但还不是动态数组，在运行期内不能再改变，受制于其它因素，它只是提供了一个部分解决方案。

第十章 动态数组

当写下这个题目的时候，笔者心里其实非常犯难。因为从本质上来说，本章想阐述的内容与题目所宣示的概念，其实是不一样的。在编程中，我们常常要处理一段长度未知的数据，而且，运行过程中长度可能会发生变化，现行的 C/C++标准没有提供在栈段和数据段内存中的实现，只提供堆中的实现，例如可以象下面代码那样在堆中分配一段内存，以处理一组长度不确定的整数：

```
int *p = ( int* )malloc( n * sizeof( int ) );
```

现在我们常常将这段堆内存称为“动态数组”。这正确吗？数组是一个高层概念，是 C/C++对象模型及类型系统的重要组成。一个对象欲成为一个数组，引用此对象的表达式或标识符必须具有高层的数组类型，但这段内存没有任何数组类型的引用，只有一个指向它的指针，因此，这段内存不是 C/C++高层语义上的数组。虽然 p 可以使用下标运算符访问内存块中的数据，但其实只不过得益于下标运算符的指针性质（如第四章所述）而已。一个真正的动态数组，不仅长度在运行期内可变，还需要具备数组类型的抽象，这要求语言规则的支持，这些条件是 p 所不具备的。但是，真正的动态数组的实现也不容易，往往受到效率等多重因素的制约，即使实现了也可能需要付出很大的代价，得不偿失，正因如此，C/C++标准都没有提供对动态数组的支持。不过，这段堆内存被称为“动态数组”多年来已经习惯成自然了，笔者没有为其重新命名的技术能力和资历，也就只有随波逐流，暂且也称之为动态数组吧，重要的是明白两者本质的不同。

鉴于动态数组不是真正的受 C/C++规则支持的动态数组，因此需要通过指针对数组内部各维地址进行构造，整个数组才能使用下标运算符。这就使动态数组的内部构造分成两部分，一部分叫数据存储区，用来保存真正的数组元素，另一部分叫中间地址缓冲区，保存数组内部各维的中间地址。

根据数据存储区的空间连续性，可以将动态数组分成两大类，一类是具有连续存储空间

的动态数组，另一类是非连续存储空间的动态数组。笔者分别将它们称为连续动态数组和离散动态数组。

离散动态数组是最简单的动态数组，因为无须考虑数据在哪里存储，只需要动态分配就行了，同时中间地址不需要或只需要简单计算就可以得出。例如一个二维离散动态数组可以这样构造：

```
int **p = ( int** )malloc( m * sizeof( int* ) );
for( i = 0; i < m; ++i )
        p[i] = ( int* )malloc( n * sizeof( int ) );
```

上述代码中，p 指向中间地址缓冲区，保存第一维各元素的首地址，p[i]指向数据存储区，这个存储区是不连续的。释放空间时需要这样进行：

```
for( i = 0; i < m; ++i )
        free( p[i] );
free( p );
```

离散动态数组是先构造好中间地址缓冲区，再构造数据存储区，这是造成数据空间不连续的原因，虽然构造过程简单，但非连续性带来很多缺点。一是不利于数组内部 的直接寻址，例如通过数据区首地址计算元素地址；二是当需要对数组长度进行改变时，过程复杂；三是空间的释放需要对中间地址缓冲区重新遍历。但其实，完全 可以先构造数据存储区，再构造中间地址缓冲区，这种方法使连续数据存储空间有了可能，而且，连续动态数组不会带来离散动态数组那些缺点。下面是构造连续动 态数组的示例：

```
int *p = ( int* )malloc( m * n * sizeof( int ) );
int **q = ( int** )malloc( m * sizeof( int* ) );
for( i = 0; i < m; ++i )
        q[i] = p + i * n;
```

首先 p 分配 m*n 个 int 数据的存储区，再由 q 根据这段空间构造中间地址。现在，不仅可以通过 q[m][n]使用这个数组，还可以直接通过 p 和下标运算符访问数组的元素。释放空间的时候直接释放 p 和 q 就行了，需要改变数组长度的话，只须重新分配 p 指向的空间，再重新构造一下中间地址缓冲区，例如将上述 m*n 个 int 元素的数组改为 k*j 个 int 元素，可以这样做：

```
int *p = ( int* )realloc( p, k * j * sizeof( int ) );
int **q = ( int** )realloc( q, k * sizeof( int* ) );
for( i = 0; i < k; ++i )
        q[i] = p + i * j;
```

而离散动态数组就必须先动态分配好 k*j 个 int 的新空间，然后把旧数据都复制过去，再释放旧空间，整个过程比连续空间麻烦得多。

连续动态数组不仅可以使用堆中的空间，还可以在栈段和数据段中构造，只要在栈或数据段的数组对象中重新构造中间地址即可，例如：

```
double a[100];
double **p = ( double** )malloc( 5 * sizeof( double* ) );
for( i = 0; i < 5; ++i )
        p[i] = a + i * 20;
```

这样就把一维 double 数组 a 的空间重新在逻辑上改造成了一个二维数组 p[5][20]，注意重新构造的动态数组的长度不能超出 a 的空间，否则结果是不确定的，是危险的。

上述例子在一个一维数组上构造了一个二维数组，维度发生了变化，这说明连续动态数组不仅可以方便地改变长度，还可以方便地改变维度。当目标维度可变时，中间地址的构造需要使用递归算法。笔者的博客中就提供了一个维度可变的数组 ADT 的例子。

要注意的是，动态数组的中间地址不具数组类型。例如上述动态数组 q[m][n]的第一维 q[m]类型依然是 int*，而一个数组对象 int a[m][n]的第一维 a[m]的类型是数组类型 int[n]。

综合来看，由于连续动态数组的优点比离散动态数组多得多，在编程实践中应优先使用连

续动态数组。