曾经碰到过让你迷惑不解、类似于 int * (* (*fp1) (int)) [10];这样的变量声明吗？
本文将由易到难，一步一步教会你如何理解这种复杂的 C/C++声明。

我们将从每天都能碰到的较简单的声明入手，然后逐步加入 const 修饰符和 typedef，还有函数指针，最后介绍一个能够让你准确地理解任何 C/C++声明的“右左法则”。

需要强调一下的是，复杂的 C/C++声明并不是好的编程风格；我这里仅仅是教你如何去理解这些声明。注意：为了保证能够在同一行上显示代码和相关注释，本文最好在至少 1024x768 分辨率的显示器上阅读。

让我们从一个非常简单的例子开始，如下：

int n;

这个应该被理解为“declare n as an int”（n 是一个 int 型的变量）。接下去来看一下指针变量，如下：

int *p;

这个应该被理解为“declare p as an int *”（p 是一个 int *型的变量），或者说 p 是一个指向一个 int 型变量的指针。我想在这里展开讨论一下：我觉得在声明一个指针（或引用）类型的变量时，最好将*（或&）写在紧靠变量之前，而不是紧跟基本类型之后。这样可以避免一些理解上的误区，比如：

再来看一个指针的指针的例子：

char **argv;

理论上，对于指针的级数没有限制，你可以定义一个浮点类型变量的指针的指针的指针的指针，再来看如下的声明：

int RollNum[30][4];

int (*p)[4]=RollNum;

int *q[5];

这里，p 被声明为一个指向一个 4 元素（int 类型）数组的指针，而 q 被声明为一个包含 5 个元素（int 类型的指针）的数组。另外，我们还可以在同一个声明中混合实用*和&，如下：

int **p1;

// p1 is a pointer   to a pointer   to an int.

```
int *&p2;

// p2 is a reference to a pointer   to an int.

int &*p3;

// ERROR: Pointer   to a reference is illegal.

int &&p4;

// ERROR: Reference to a reference is illegal.
```

注：p1 是一个 int 类型的指针的指针；p2 是一个 int 类型的指针的引用；p3 是一个 int 类型引用的指针（不合法!）；p4 是一个 int 类型引用的引用（不合法!）。

const 修饰符

当你想阻止一个变量被改变，可能会用到 const 关键字。在你给一个变量加上 const 修饰符的同时，通常需要对它进行初始化，因为以后的任何时候你将没有机会再去改变它。例如：

```
const int n=5;

int const m=10;
```

上述两个变量 n 和 m 其实是同一种类型的——都是 const int（整形恒量）。因为 C++标准规定，const 关键字放在类型或变量名之前等价的。我个人更喜欢第一种声明方式，因为它更突出了 const 修饰符的作用。当 const 与指针一起使用时，容易让人感到迷惑。例如，我们来看一下下面的 p 和 q 的声明：

```
const int *p;

int const *q;
```

他们当中哪一个代表 const int 类型的指针（const 直接修饰 int），哪一个代表 int 类型的 const 指针（const 直接修饰指针）？实际上，p 和 q 都被声明为 const int 类型的指针。而 int 类型的 const 指针应该这样声明：

```
int * const r= &n;

// n has been declared as an int
```

这里，p 和 q 都是指向 const int 类型的指针，也就是说，你在以后的程序里不能改变*p 的值。

而 r 是一个 const 指针，它在声明的时候被初始化指向变量 n（即 r=&n;）之后，r 的值将不再允许被改变（但*r 的值可以改变）。

组合上述两种 const 修饰的情况，我们来声明一个指向 const int 类型的 const 指针，如下：

```
const int * const p=&n

// n has been declared as const int
```

下面给出的一些关于 const 的声明，将帮助你彻底理清 const 的用法。不过请注意，下面的一些声明是不能被编译通过的，因为他们需要在声明的同时进行初始化。为了简洁起见，我忽略了初始化部分；因为加入初始化代码的话，下面每个声明都将增加两行代码。

```
char ** p1;

//     pointer to     pointer to     char

const char **p2;

//     pointer to     pointer to const char

char * const * p3;

//     pointer to const pointer to     char

const char * const * p4;

//     pointer to const pointer to const char

char ** const p5;

// const pointer to     pointer to     char

const char ** const p6;

// const pointer to     pointer to const char

char * const * const p7;

// const pointer to const pointer to     char
```

const char * const * const p8;

// const pointer to const pointer to const char

注：p1 是指向 char 类型的指针的指针；p2 是指向 const char 类型的指针的指针；p3 是指向 char 类型的 const 指针；p4 是指向 const char 类型的 const 指针；p5 是指向 char 类型的指针的 const 指针；p6 是指向 const char 类型的指针的 const 指针；p7 是指向 char 类型 const 指针的 const 指针；p8 是指向 const char 类型的 const 指针的 const 指针。

typedef 的妙用

typedef 给你一种方式来克服“*只适合于变量而不适合于类型”的弊端。你可以如下使用 typedef：

typedef char * PCHAR;

PCHAR p,q;

这里的 p 和 q 都被声明为指针。（如果不使用 typedef，q 将被声明为一个 char 变量，这跟我们的第一眼感觉不太一致！）下面有一些使用 typedef 的声明，并且给出了解释：

typedef char * a;

// a is a pointer to a char

typedef a b();

// b is a function that returns

// a pointer to a char

typedef b *c;

// c is a pointer to a function

// that returns a pointer to a char

typedef c d();

```
// d is a function returning

// a pointer to a function

// that returns a pointer to a char

typedef d *e;

// e is a pointer to a function

// returning a pointer to a

// function that returns a

// pointer to a char

e var[10];

// var is an array of 10 pointers to

// functions returning pointers to

// functions returning pointers to chars.
```

typedef 经常用在一个结构声明之前，如下。这样，当创建结构变量的时候，允许你不使用关键字 struct（在 C 中，创建结构变量时要求使用 struct 关键字，如 struct tagPOINT a；而在 C++中，struct 可以忽略，如 tagPOINT b）。

```
typedef struct tagPOINT

{

  int x;

  int y;

}POINT;



POINT p;
```

函数指针

函数指针可能是最容易引起理解上的困惑的声明。函数指针在 DOS 时代写 TSR 程序时用得最多；在 Win32 和 X-Windows 时代，他们被用在需要回调函数的场合。当然，还有其它很多地方需要用到函数指针：虚函数表，STL 中的一些模板，Win NT/2K/XP 系统服务等。让我们来看一个函数指针的简单例子：

int (*p)(char);

这里 p 被声明为一个函数指针，这个函数带一个 char 类型的参数，并且有一个 int 类型的返回值。另外，带有两个 float 类型参数、返回值是 char 类型的指针的指针的函数指针可以声明如下：

char ** (*p)(float, float);

那么，带两个 char 类型的 const 指针参数、无返回值的函数指针又该如何声明呢？参考如下：

void * (*a[5])(char * const, char * const);

“右左法则”是一个简单的法则，但能让你准确理解所有的声明。这个法则运用如下：从最内部的括号开始阅读声明，向右看，然后向左看。当你碰到一个括号时就调转阅读的方向。括号内的所有内容都分析完毕就跳出括号的范围。这样继续，直到整个声明都被分析完毕。

对上述“右左法则”做一个小小的修正：当你第一次开始阅读声明的时候，你必须从变量名开始，而不是从最内部的括号。

下面结合例子来演示一下“右左法则”的使用。

int * (* (*fp1) (int)) [10];

阅读步骤：

1. 从变量名开始——fp1

2. 往右看，什么也没有，碰到了)，因此往左看，碰到一个*——一个指针

3. 跳出括号，碰到了(int)——一个带一个 int 参数的函数

4. 向左看，发现一个*——（函数）返回一个指针

5. 跳出括号，向右看，碰到[10]——一个 10 元素的数组

6. 向左看，发现一个*——指针

7. 向左看，发现 int——int 类型

总结：fp1 被声明成为一个函数的指针,该函数返回指向指针数组的指针.

再来看一个例子：

int *(*(*arr[5])())();

阅读步骤：

1. 从变量名开始——arr

2. 往右看，发现是一个数组——一个 5 元素的数组

3. 向左看，发现一个*——指针

4. 跳出括号，向右看，发现()——不带参数的函数

5. 向左看，碰到*——（函数）返回一个指针

6. 跳出括号，向右发现()——不带参数的函数

7. 向左，发现*——（函数）返回一个指针

8. 继续向左，发现 int——int 类型

还有更多的例子：

float (* (*b()) [])();

// b is a function that returns a

// pointer to an array of pointers

// to functions returning floats.

```
void * ( *c) ( char, int (*)());

// c is a pointer to a function that takes

// two parameters:

// a char and a pointer to a

// function that takes no

// parameters and returns

// an int

// and returns a pointer to void.

void ** (*d) (int &,

char **(*)(char *, char **));

// d is a pointer to a function that takes

// two parameters:

// a reference to an int and a pointer

// to a function that takes two parameters:

// a pointer to a char and a pointer

// to a pointer to a char

// and returns a pointer to a pointer

// to a char

// and returns a pointer to a pointer to void

float ( * ( * e[10])

   (int &) ) [5];

// e is an array of 10 pointers to
```

// functions that take a single

// reference to an int as an argument

// and return pointers to

// an array of 5 floats.

补充：

C 语言所有复杂的指针声明，都是由各种声明嵌套构成的。如何解读复杂指针声明呢？右左法则是一个既著名又常用的方法。不过，右左法则其实并不是 C 标准里面的内容，它是从 C 标准的声明规定中归纳出来的方法。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，两者可以说是相反的。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 语言的数组名是一个右值，它不能作为左值来接收另一个数组，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，因为数组的元素除了类型必须一样外，每个元素所占用的内存空间也必须相同，显然函数是无法达到这个要求的，即使函数的类型一样，但函数所占用的空间通常是不相同的。

作为练习，下面列几个复杂指针声明给读者自己来解析，答案放在第十章里。

```
int (*(*func)[5][6])[7][8];

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

int (*(*func[7][8][9])(int*))[5];
```

实际当中，需要声明一个复杂指针时，如果把整个声明写成上面所示的形式，对程序可读性是一大损害。应该用 typedef 来对声明逐层分解，增强可读性，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。