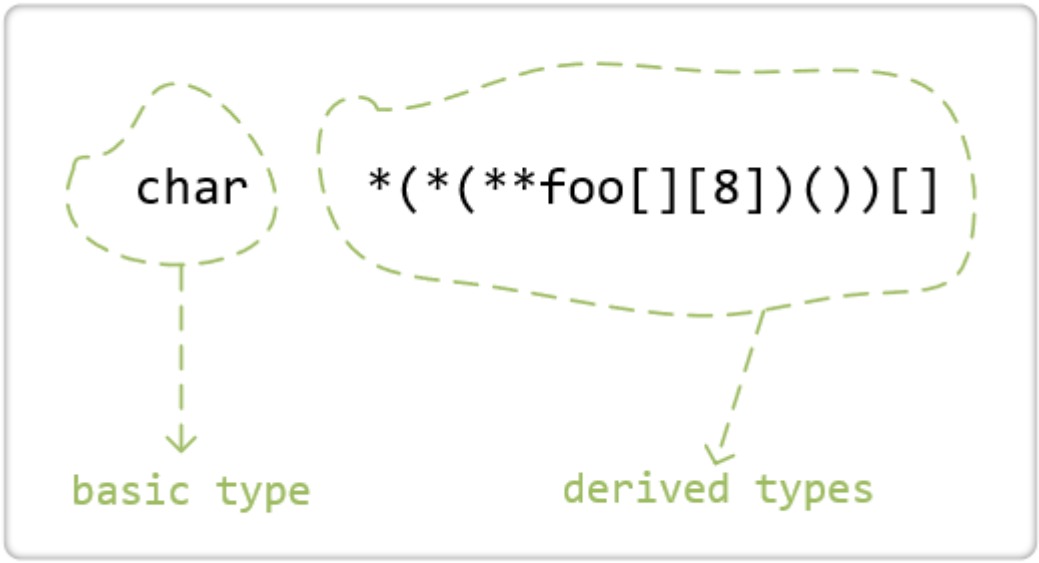
对于一些复杂的声明式，我们往往很难理解它代表的含义，

```
int (*(*(*foo)(int*))[5])(int*); /* complex and difficult to understand */
```

“右左法则“是前辈们总结出的一套分析规则，借助它，可以准确快速地分析出上式的具体含义。

一：声明式的组成

一个声明式可以分为两部分，“basic type”和“derived types”，如下，



对于“derived types”，它主要由以下几部分组成，

`foo`，标识符。

`*`，**pointer to...**，指针符号。

`[]`，**array of...**，表示数组的一对方括号。注意方括号内也可以有个数字，例如`[10]`，那么它的含义就是"array of 10..."。

`()`，**function returning...**，表示函数名后面包住参数的一对括号。注意括号内也可以有参数，例如`(int*)`，那么它的含义就是“function with parameter 'int*' returning...”。

二：右左法则

根据运算符优先级，"array of" `[]`和"function returning" `()`的优先级比"pointer to" `*`高，因此定义的规则如下：

1. 总是以标识符为开头，

   foo is ...

2. 总是以“basic type”为结尾，

foo is ... **char**

3. 中间部分比较复杂，但有一个规律，

   “go right when you can, go left when you must”

接下来我们就分别以一易一难两个例子具体讲下这个“右左法则”。

三：简单的例子

```
long **foo[7];
```

我们一步一步地按照上面的“右左法则”来，

- **long** * * **foo** [7]

  以标识符为开头，以“basic type”为结尾，

  foo is ... long

- ~~long~~ * * ~~foo~~ **[7]**

  根据“右左法则”的“go right when you can“，下一步是分析”array of 7...“`[7]`，

  foo is array of 7 ... long

- ~~long~~ * ***** ~~foo [7]~~

  已走到右边的尽头，故“go left when you must”，下一步分析指针符号`*`，

  foo is array of 7 pointer to ... long

- ~~long~~ ***** ~~* foo [7]~~

  继续“go left when you must”，依旧是一个指针，

  foo is array of 7 pointer to pointer to long

- ~~long * * foo [7]~~

  全部分析完毕，完整的含义为：*foo is array of 7 pointer to pointer to long*，翻译为：foo是一个长度为7的数组，这个数组的元素是指向long型指针的指针。

四：复杂的例子

```
int (*(*(*foo)(int*))[5])(int*);
```

- **int** (* (* (* **foo**) (int*)) [5]) (int*)

  以标识符为开头，以“basic type”为结尾，

  foo is ... int

- ~~int~~ (* (* (***** ~~foo~~) (int*)) [5]) (int*)

  走到括号尾，故“go left when you must”，下一步分析指针符号`*`，

  foo is pointer to ... int

- ~~int~~ (* (* ~~(* foo)~~ **(int*)**) [5]) (int*)

  一对括号内的内容分析完，继续“go right when you can“，下一步分析`(int*)`，

  foo is pointer to function with parameter 'int' returning ... int*

- ~~int~~ (* (***** ~~(* foo) (int*)~~) [5]) (int*)

  走到括号尾，故“go left when you must”，下一步分析指针符号 `*`，

  foo is pointer to function with parameter 'int' returning pointer to … int*

- ~~int~~ (* ~~(* (* foo) (int*))~~ **[5]**) (int*)

  此时可以“go right when you can“，下一步分析 `[5]`，

  foo is pointer to function with parameter 'int' returning pointer to array of 5 … int*

- ~~int~~ (***** ~~(* (* foo) (int*)) [5]~~) (int*)

  走到括号尾，故“go left when you must”，下一步分析指针符号，

  foo is pointer to function with parameter 'int' returning pointer to array of 5 pointer to … int*

- ~~int (* (* (* foo) (int*)) [5])~~ **(int*)**

  此时可以“go right when you can“，下一步分析 `(int*)`，

  foo is pointer to function with parameter 'int' returning pointer to array of 5 pointer to function with parameter 'int*' returning int*

- ~~int (* (* (* foo) (int*)) [5]) (int*)~~

  全部分析完毕，完整的含义为：*foo is pointer to function with parameter 'int*' returning pointer to array of 5 pointer to function with parameter 'int*' returning int*，翻译为：foo是一个函数指针，这个函数的参数是“int*”，返回的是一个指向长度为5的数组的数组指针，这个数组的元素也是函数指针，参数也是“int*”，返回的是“int”。

五：抽象声明式（Abstract Declarators）

抽象声明式（Abstract Declarators）通常用作数据类型转换和作为 `sizeof` 的参数，与上面几个声明式不同的是，抽象声明式缺少**标识符**，

```
int (*(*)())()
```

“右左法则”都是从标识符开始分析的，那么问题来了，抽象声明式没有标识符，该从哪里开始呢？**找到它应该在的位置。**

我们仔细地观察，会发现，不管是什么声明式，标识符满足下面几条规律：

1. 都在“pointer to” `*` 的右边；
2. 都在“array of” `[]` 的左边；
3. 都在“function returning” `()` 的左边；
4. 都在括号（除了表示“function returning”的括号）里边。

根据这几条规律，我们再来看看上面的抽象声明式。根据规律1和规律3，可以锁定大致范围，

~~**int (* (* •) • ( )) ( )**~~

其中“•”这个点就代表标识符应该在的位置，再根据规律4，最终锁定在左边那个“•”的位置，那么这个抽象声明式我们可以看成这样，

```
int (*(*foo)())()
```

如此，我们就可以很容易地知道它的含义了，*foo is pointer to function returning pointer to function returning int*。

六：调用约定（Calling Conventions）

在Windows平台上的Win32编程中，我们经常会碰到一些调用约定符号`__cdecl`，`__fastcall`和`__stdcall`等等，

```
extern int __cdecl main(int argc, char **argv);

extern BOOL __stdcall DrvQueryDriverInfo(DWORD dwMode, PVOID pBuffer, DWORD cbBuf,
PDWORD pcbNeeded);
```

这些调用约定符号是用来修饰函数的，所以若出现在声明式中，我们应当将其置于“function returning”`()`之前，比如，

```
BOOL (__stdcall *foo)();
```

它的含义为：*foo is pointer to __stdcall function returning BOOL*。

七：参考文献

- http://unixwiz.net/techtips/reading-cdecl.html
- cdecl