3.5 typedef 可以成为你的朋友

typedef 是一种有趣的声明形式：它为一种类型引入新的名字，而不是为变量分配空间。在某些方面，typedef 类似于宏文本替换——它并没有引入新类型，而是为现有类型取个新名字，但它们之间存在一个关键性的区别，容我稍后解释。

如果现在回过头去看看“声明是如何形成的”那一节，会发现 typedef 关键字可以是一个常规声明的一部分，可以出现在靠近声明开始部分的任何地方。事实上，typedef 的格式与变量声明完全一样，只是多了这个关键字，向你提醒它的实质。

由于 typedef 看上去跟变量声明完全一样，它们读起来也是一样的。前面一节描述的分析技巧也同样适用于 typedef。普通的声明表示“这个名字是一个指定类型的变量”，而 typedef 关键字并不创建一个变量，而是宣称“这个名字是指定类型的同义词”。

一般情况下，typedef 用于简洁地表示指向其他东西的指针。典型的例子是 signal()原型的声明。signal()是一种系统调用，用于通知运行时系统，当某种特定的“软件中断”发生时调用特定的程序。它的真正名称应该是“Call_that_routine_when_this_interrupt_ comes_in（当该中断发生时调用那个程序）”。你调用 signal()，并通过参数传递告诉它中断的类型以及用于处理中断的程序。在 ANSI C 标准中，signal()的声明如下：

```
void (*signal(int sig, void(*func)(int)))(int);
```

让我们运用刚刚掌握的技巧来分析这个声明，会发现它的意思如下：

```
void(*signal(           ))(int);
```

signal 是一个函数（具有一些令人胆战心惊的参数），它返回一个函数指针，后者所指向的函数接受一个 int 参数并返回 void。其中一个恐怖的参数是其本身：

```
void(*func)(int);
```

它表示一个函数指针，所指向的函数接受一个 int 参数，返回值是 void。现在，让我们看一下怎样用 typedef 来“代表”通用部分，从而进行简化。

```
typedef  void(*ptr_to_func) (int);
/*  它表示 ptr_to_func 是一个函数指针，该函数
 *  接受一个 int 参数，返回值为 void。
 */

ptr_to_func signal(int, ptr_to_func);
/*  它表示 signal 是一个函数，它接受两个参数，
 *  其中一个是 int，另一个是 ptr_to_func，返回
 *  值是 ptr_to_func。
 */
```

然而，说到 typedef 就不能不说一下它的缺点。它同样具有与其他声明一样的混乱语法，同样可以把几个声明器塞到一个声明中去。对于结构，除了可以在书写时省掉 struct 关键字之外，typedef 并不能提供显著的好处，而少写一个 struct 其实并没有多大帮助。在任何 typedef 声明中，甚至不必把 typedef 放在声明的开始位置。

操作声明器的一些提示

不要在一个 typedef 中放入几个声明器，如下所示：

```
typedef int *ptr, (fun)(), arr[5];
/*   ptr是“指向int的指针”类型，
 *   fun是“指向返回值为int的函数的指针”类型
 *   arr是“长度为5的int型数组”类型
 */
```

千万不要把 typedef 嵌到声明的中间部分，如下所示：

```
unsigned const long typedef int volatile *kumquat;
```

typedef 为数据类型创建别名，而不是创建新的数据类型，可以对任何类型进行 typedef 声明。

```
typedef int (*array_ptr)[100];
```

应该只对所希望的变量类型进行 typedef 声明，为变量类型取一个喜欢的别名。关键字 typedef 应该如前所述出现在声明的开始位置。在同一个代码块中，typedef 引入的名字不能与其他标识符同名。

3.6 typedef int x[10]和#define x int[10]的区别

前面已经提到过，在 typedef 和宏文本替换之间存在一个关键性的区别。正确思考这个问题的方法就是把 typedef 看成是一种彻底的“封装”类型——在声明它之后不能再往里面增加别的东西。它和宏的区别体现在两个方面。

首先，可以用其他类型说明符对宏类型名进行扩展，但对 typedef 所定义的类型名却不能这样做。如下所示：

```
#define peach int
unsigned peach i; /* 没问题 */


typedef int banana;
unsigned banana i; /* 错误！ 非法 */
```

其次，在连续几个变量的声明中，用 typedef 定义的类型能够保证声明中所有的变量均为同一种类型，而用#define 定义的类型则无法保证。如下所示：

```
#define int_ptr int *
int_ptr chalk, cheese;
```

经过宏扩展，第二行变为：

```
int * chalk, cheese;
```

这使得 chalk 和 cheese 成为不同的类型，就好象是辣椒酱与细香葱的区别：chalk 是一个指向 int 的指针，而 cheese 则是一个 int。相反，下面的代码中：

```
typedef char * char_ptr;
char_ptr Bentley, Rolls_Royce;
```

Bentley 和 Rolls_Royce 的类型依然相同。虽然前面的类型名变了，但它们的类型相同，都是指向 char 的指针。

操作 typedef 的提示

不要为了方便起见对结构使用 typedef。

这样做惟一的好处是能使你不必书写“struct”关键字，但这个关键字可以向你提示一些信息，你不应该把它省掉。

typedef 应该用在：

- 数组、结构、指针以及函数的组合类型。
- 可移植类型。比如当你需要一种至少 20 比特的类型时，可以对它进行 typedef 操作 typedef 的提示声明。这样，当把代码移植到不同的平台时，要选择正确的类型如 short,int,long 时，只要在 typedef 中进行修改就可以了，无需对每个声明都加以修改。
- typedef 也可以为后面的强制类型转换提供一个简单的名字，如：

```
typedef int (*ptr_to_int_fun)(void);
  char * p;  ...
             = (ptr_to_int_fun) p;
```

应该始终在结构的定义中使用结构标签，即使它并非必须。这种做法可以使代码更为清晰。

typedef工具是一个高级数据特性，利用typedef可以为某一类型自定义名称。这方面与#define类似，但是两者有3处不同：

与#define不同，typedef创建的符号名只受限于类型，不能用于值。

typedef由编译器解释，不是预处理器。

在其受限范围内，typedef比#define更灵活。

使用typedef的第2个原因是：typedef常用于给复杂的类型命名。例如，下面的声明：

typedef char (* FRPTC ()) [5];

把FRPTC声明为一个函数类型，该函数返回一个指针，该指针指向内含5个char类型元素的数组（参见下一节的讨论）。

使用typedef时要记住，typedef并没有创建任何新类型，它只是为某个已存在的类型增加了一个方便使用的标签。以前面的STRING为例，这意味着我们创建的STRING类型变量可以作为实参传递给以指向char指针作为形参的函数。