### 关于typedef的用法总结

### 在C还是C++代码中，typedef都使用的很多，在C代码中尤其是多。typedef与#define有些相似，其实是不同的，特别是在一些复杂的用法上，看了网上一些C/C++的学习者的博客，其中有一篇关于typedef的总结还是很不错，由于总结的很好，我就不加修改的引用过来了，加上自己的一个分析。

## ****基本定义：****

typedef为C语言的关键字，作用是为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型（int,char等）和自定义的数据类型（struct等）。 在编程中使用typedef目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。

## ****用途一：与#define的区别****

typedef 行为有点像 #define 宏，用其实际类型替代同义字。不同点是 typedef 在编译时被解释，因此让编译器来应付超越预处理器能力的文本替换。

## ****用途二：减少错误****

定义一种类型的别名，而不只是简单的宏替换。可以用作同时声明指针型的多个对象。比如：

char\* pa, pb; // 这多数不符合我们的意图，它只声明了一个指向字符变量的指针，

// 和一个字符变量；

以下则可行：

typedef char\* PCHAR;

PCHAR pa, pb;

这种用法很有用，特别是char\* pa, pb的定义，初学者往往认为是定义了两个字符型指针，其实不是，而用typedef char\* PCHAR就不会出现这样的问题，减少了错误的发生。

## ****用途三：直观简洁****

用在旧的C代码中，帮助struct。以前的代码中，声明struct新对象时，必须要带上struct，即形式为： struct 结构名对象名，如：

struct tagPOINT1

{

int x;

int y;

};

struct tagPOINT1 p1;

而在C++中，则可以直接写：结构名对象名，即：tagPOINT1 p1;

typedef struct tagPOINT

{

int x;

int y;

}POINT;

POINT p1; // 这样就比原来的方式少写了一个struct，比较省事，尤其在大量使用的时候,或许，在C++中，typedef的这种用途二不是很大，但是理解了它，对掌握以前的旧代码还是有帮助的，毕竟我们在项目中有可能会遇到较早些年代遗留下来的代码。

## 用途四：平台无关性

用typedef来定义与平台无关的类型。

typedef 有另外一个重要的用途，那就是定义机器无关的类型，例如，你可以定义一个叫 REAL 的浮点类型，在目标机器上它可以获得最高的精度：

　　typedef long double REAL;

在不支持 long double 的机器上，该 typedef 看起来会是下面这样：

　　typedef double REAL;

并且，在连 double 都不支持的机器上，该 typedef 看起来会是这样：

　　typedef float REAL;

也就是说，当跨平台时，只要改下 typedef 本身就行，不用对其他源码做任何修改。

标准库就广泛使用了这个技巧，比如size\_t。另外，因为typedef是定义了一种类型的新别名，不是简单的字符串替换，所以它比宏来得稳健。

## ****用途五：掩饰复合类型****

typedef 还可以掩饰复合类型，如指针和数组。

例如，你不用像下面这样重复定义有 81 个字符元素的数组：

　　char line[81];

　　char text[81];

定义一个 typedef，每当要用到相同类型和大小的数组时，可以这样：

　　typedef char Line[81];

此时Line类型即代表了具有81个元素的字符数组，使用方法如下：

　　Line text, secondline;

　　getline(text);

同样，可以象下面这样隐藏指针语法：

　　typedef char \* pstr;

　　int mystrcmp(pstr, pstr);

这里将带我们到达第一个 typedef 陷阱。标准函数 strcmp()有两个‘ const char \*'类型的参数。因此，它可能会误导人们象下面这样声明 mystrcmp()：

　　int mystrcmp(const pstr, const pstr);

用GNU的gcc和g++编译器，是会出现警告的，按照顺序，‘const pstr'被解释为‘char\* const‘（一个指向 char 的指针常量），两者表达的并非同一意思。为了得到正确的类型，应当如下声明：

　　typedef const char\* pstr;

## ****用途六：代码简化****

代码简化。为复杂的声明定义一个新的简单的别名。方法是：在原来的声明里逐步用别名替换一部分复杂声明，如此循环，把带变量名的部分留到最后替换，得到的就是原声明的最简化版。举例：

 原声明：

void (\*b[10]) (void (\*)());

变量名为b，先替换右边部分括号里的，pFunParam为别名

typedef void (\*pFunParam)();

再替换左边的变量b，pFunx为别名二：

typedef void (\*pFunx)(pFunParam);

原声明的最简化版：

pFunx b[10];

原声明：

doube(\*)() (\*e)[9];

变量名为e，先替换左边部分，pFuny为别名一：

typedef double(\*pFuny)();

再替换右边的变量e，pFunParamy为别名二

typedef pFuny (\*pFunParamy)[9];

原声明的最简化版：

pFunParamy e;

理解复杂声明可用的“右左法则”：从变量名看起，先往右，再往左，碰到一个圆括号就调转阅读的方向；括号内分析完就跳出括号，还是按先右后左的顺序，如此循环，直到整个声明分析完。举例：

int (\*func)(int \*p);

首先找到变量名func，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，又遇到圆括号，这说明(\*func)是一个函数，所以func是一个指向这类函数的指针，即函数指针，这类函数具有int\*类型的形参，返回值类型是int。

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

func右边是一个[]运算符，说明func是具有5个元素的数组；func的左边有一个\*，说明func的元素是指针（注意这里的\*不是修饰func，而是修饰func[5]的，原因是[]运算符优先级比\*高，func先跟[]结合）。跳出这个括号，看右边，又遇到圆括号，说明func数组的元素是函数类型的指针，它指向的函数具有int\*类型的形参，返回值类型为int。

## ****用途七：typedef 和存储类关键字（storage class specifier）****

这种说法是不是有点令人惊讶，typedef 就像 auto，extern，mutable，static，和 register 一样，是一个存储类关键字。这并不是说 typedef 会真正影响对象的存储特性；它只是说在语句构成上，typedef 声明看起来象 static，extern 等类型的变量声明。下面将带到第二个陷阱：

　　typedef register int FAST\_COUNTER; // 错误

　　编译通不过。问题出在你不能在声明中有多个存储类关键字。因为符号 typedef 已经占据了存储类关键字的位置，在 typedef 声明中不能用 register（或任何其它存储类关键字）。