为什么很多人编程喜欢用typedef?

嵌入式ARM 2022-03-08 12:00

摘要：不同的项目，有不同的代码风格，当然也有不同的代码“癖好”。代码看的多了，你就会发现：有的代码喜欢用宏，有的代码喜欢用typedef。那么，使用typedef到底有哪些好处呢？为什么很多人都喜欢用它？

1、typedef 的基本使用

1.1 typedef与结构体的结合使用

typedef是C语言的一个关键字，用来给某个类型起个别名，也就是给C语言中已经存在的一个类型起一个新名字。大家在阅读代码的过程中，会经常见到 typedef 与结构体、联合体、枚举、函数指针声明结合使用。比如下面结构体类型的声明和使用：

```
struct student
{
  char name[20];
  int  age;
  float score;
};
struct student stu = {"wit", 20, 99};
```

在C语言中定义一个结构体变量，我们通常的写法是：

```
struct 结构体名 变量名;
```

前面必须有一个struct关键字打前缀，编译器才会理解你要定义的对象是一个结构体变量。而在C++语言中，则不需要这么做，直接使用：结构体名 变量名就可以了

```
struct student
{
  char name[20];
  int age;
  float score;
};
int main (void)
{
  student stu = {"wit", 20, 99};
  return 0;
}
```

如果我们使用typedef，就可以给student声明一个别名student_t和一个结构体指针类型student_ptr，然后就可以直接使用student_t类型去定义一个结构体变量，不用再写struct，这样会显得代码更加简洁。

```
#include <stdio.h>
```

```
typedef struct student
{
  char name[20];
  int  age;
  float score;
}student_t, *student_ptr;

int main (void)
{
  student_t   stu = {"wit", 20, 99};
  student_t  *p1 = &stu;
  student_ptr p2 = &stu;
  printf ("name: %s\n", p1->name);
  printf ("name: %s\n", p2->name);
  return 0;
}
程序运行结果:
wit
wit
```

1. 2 typedef 与数组的结合使用

typedef除了与结构体结合使用外，还可以与数组结合使用。定义一个数组，通常我们使用int array[10];即可。我们也可以使用typedef先声明一个数组类型，然后再使用这个类型去定义一个数组。

```
typedef int array_t[10];
array_t array;
```

```
int main (void)
{
  array[9] = 100;
  printf ("array[9] = %d\n", array[9]);
  return 0;
}
```

在上面的demo程序中，我们声明了一个数组类型array_t，然后再使用该类型定义一个数组array，这个array效果其实就相当于：int array[10]。

1.3 typedef 与指针的结合使用

```
typedef char * PCHAR;
int main (void)
{
    //char * str = "学嵌入式";
  PCHAR str = "学嵌入式";
  printf ("str: %s\n", str);
  return 0;
}
```

在上面的demo程序中，PCHAR 的类型是char *，我们使用PCHAR类型去定义一个变量str，其实就是一个char *类型的指针。

1.4 typedef与函数指针的结合使用

定义一个函数指针，我们通常采用下面的形式：

```
int (*func)(int a, int b);
```

我们同样可以使用typedef声明一个函数指针类型：func_t

```
typedef int (*func_t)(int a, int b);
func_t fp;  // 定义一个函数指针变量
```

写个简单的程序测试一下，运行OK：

```
typedef int (*func_t)(int a, int b);
int sum (int a, int b)
{
  return a + b;
}
int main (void)
{
  func_t fp = sum;
  printf ("%d\n", fp(1,2));
  return 0;
}
```

为了增加程序的可读性，我们经常在代码中看到下面的声明形式：

```
typedef int (func_t)(int a, int b);
func_t *fp = sum;
```

函数都是有类型的，我们使用typedef给函数类型声明一个新名称：func_t。这样声明的好处是：即使你没有看到func_t的定义，也能够清楚地知道fp是一个函数指针，代码的可读性比上面的好。

1.5 typedef与枚举的结合使用

```
typedef enum color
{
  red,
  white,
  black,
  green,
  color_num,
} color_t;

int main (void)
{
  enum color color1 = red;
  color_t    color2 = red;
  color_t color_number = color_num;
  printf ("color1: %d\n", color1);
  printf ("color2: %d\n", color2);
  printf ("color num: %d\n", color_number);
  return 0;
}
```

枚举与typedef的结合使用方法跟结构体类似：可以使用typedef给枚举类型color声明一个新名称color_t，然后使用这个类型就可以直接定义一个枚举变量。

2、使用typedef的优势

不同的项目，有不同的代码风格，也有不同的代码“癖好”。看得代码多了，你会发现：有的代码喜欢用宏，有的代码喜欢使用typedef。那么，使用typedef到底有哪些好处呢？为什么很多人喜欢用它呢？

2.1 可以让代码更加清晰简洁

```
typedef struct student
{
  char name[20];
  int  age;
  float score;
}student_t, *student_ptr;

student_t   stu = {"wit", 20, 99};
student_t  *p1 = &stu;
student_ptr p2 = &stu;
```

如示例代码所示，使用typedef，我们可以在定义一个结构体、联合、枚举变量时，省去关键字struct，让代码更加简洁。

2.2 增加代码的可移植性

C语言的int类型，我们知道，在不同的编译器和平台下，所分配的存储字长不一样：可能是2个字节，可能是4个字节，也有可能是8个字节。如果我们在代码中想定义一个固定长度的数据类型，此时使用int，在不同的平台环境下运行可能会出现问题。为了应付各种不同“脾气”的编译器，最好的办法就是使用自定义数据类型，而不是使用C语言的内置类型。

```
#ifdef PIC_16
typedef  unsigned long U32
#else
typedef unsigned int U32
#endif
```

在16位的 PIC 单片机中，int一般占2个字节，long占4个字节，而在32位的ARM环境下，int和long一般都是占4个字节。如果我们在代码中想使用一个32位的固定长度的无符号类型，可以使用上面方式声明一个U32的数据类型，在代码中你可以放心大胆地使用U32。将代码移植到不同的平台时，直接修改这个声明就可以了。

在Linux内核、驱动、BSP 等跟底层架构平台密切相关的源码中，我们会经常看到这样的数据类型，如size_t、U8、U16、U32。在一些网络协议、网卡驱动等对字节宽度、大小端比较关注的地方，也会经常看到typedef使用得很频繁。

2.3 比宏定义更好用

C语言的预处理指令#define用来定义一个宏，而typedef则用来声明一种类型的别名。typedef跟宏相比，不仅仅是简单的字符串替换，可以使用该类型同时定义多个同类型对象。

```
typedef char* PCHAR1;
#define PCHAR2 char *
```

```
int main (void)
{
  PCHAR1 pch1, pch2;
  PCHAR2 pch3, pch4;
  printf ("sizeof pch1: %d\n", sizeof(pch1));
  printf ("sizeof pch2: %d\n", sizeof(pch2));
  printf ("sizeof pch3: %d\n", sizeof(pch3));
  printf ("sizeof pch4: %d\n", sizeof(pch4));
  return 0;
}
```

在上面的示例代码中，我们想定义4个指向char类型的指针变量，然而运行结果却是：

```
sizeof pch1: 4
sizeof pch2: 4
sizeof pch3: 4
sizeof pch4: 1
```

本来我们想定义4个指向char类型的指针，但是 pch4 经过预处理宏展开后，就变成成了一个字符型变量，而不是一个指针变量。而 PCHAR1 作为一种数据类型，在语法上其实就等价于相同类型的类型说明符关键字，因此可以在一行代码中同时定义多个变量。上面的代码其实就等价于：

```
char *pch1, *pch2;
char *pch3, pch4;
```

2.4 让复杂的指针声明更加简洁

一些复杂的指针声明，如：函数指针、数组指针、指针数组的声明，往往很复杂，可读性差。比如下面函数指针数组的定义：

```
int *(*array[10])(int *p, int len, char name[]);
```

上面的指针数组定义，很多人一瞅估计就懵逼了。我们可以使用typedef优化一下：先声明一个函数指针类型func_ptr_t，接着再定义一个数组，就会更加清晰简洁，可读性就增加了不少：

```
typedef int *(*func_ptr_t)(int *p, int len, char name[]);
func_ptr_t array[10];
```

3、使用typedef需要注意的地方

通过上面的示例代码，我们可以看到，使用typedef可以让我们的代码更加简洁、可读性更强一些。但是typedef也有很多坑，稍微不注意就可能翻车。下面分享一些使用typedef需要注意的一些细节。

3.1 typedef在语法上等价于关键字

我们使用typedef给已知的类型声明一个别名，其在语法上其实就等价于该类型的类型说明符关键字，而不是像宏一样，仅仅是简单的字符串替换。举一个例子大家就明白了，比如const和类型的混合使用：当const和常见的类型(如：int、char) 一同修饰一个变量时，const和类型的位置可以互换。但是如果类型为指针，则const和指针类型不能互换，否则其修饰的变量类型就发生了变化，如常见的指针常量和常量指针：

```
char b = 10;
char c = 20;
int main (void)
{
  char const *p1 = &b; //常量指针：*p1不可变，p1可变
  char *const p2 = &b; //指针常量：*p2可变，p2不可变
  p1  = &c; //编译正常
  *p1 = 20; //error: assignment of read-only location
  p2  = &c; //error: assignment of read-only variable`p2'
  *p2 = 20; //编译正常
  return 0;
}
```

当typedef 和 const一起去修饰一个指针类型时，与宏定义的指针类型进行比较：

```
typedef char* PCHAR2;
#define PCHAR1 char *
char b = 10;
char c = 20;
int main (void)
{
  const PCHAR1 p1 = &b;
  const PCHAR2 p2 = &b;
  p1  = &c; //编译正常
  *p1 = 20; //error: assignment of read-only location
  p2  = &c; //error: assignment of read-only variable`p2'
  *p2 = 20; //编译正常
```

```
  return 0;
}
```

运行程序，你会发现跟上面的示例代码遇到相同的编译错误，原因在于宏展开仅仅是简单的字符串替换：

```
const PCHAR1 p1 = &b; //宏展开后是一个常量指针
const char * p1 = &b; //其中const与类型char的位置可以互换
```

而在使用PCHAR2定义的变量p2中，PCHAR2作为一个类型，位置可与const互换，const修饰的是指针变量p2的值，p2的值不能改变，是一个指针常量，但是*p2的值可以改变。

```
const PCHAR2 p2 = &b; //PCHAR2此时作为一个类型，与const可互换位置
PCHAR2 const p2 = &b; //该语句等价于上条语句
char * const p2 = &b; //const和PCHAR2一同修饰变量p2，const修饰的是p2!
```

3.2 typedef是一个存储类关键字

没想到吧，typedef在语法上是一个存储类关键字！跟常见的存储类关键字(如：auto、register、static、extern)一样，在修饰一个变量时，不能同时使用一个以上的存储类关键字，否则编译会报错：

```
typedef static char * PCHAR;
//error: multiple storage classes in declaration of `PCHAR'
```

3.3 typedef 的作用域

跟宏的全局性相比，typedef作为一个存储类关键字，是有作用域的。使用typedef声明的类型跟普通变量一样遵循作用域规则：包括代码块作用域、文件作用域等。

```
typedef char CHAR;

void func (void)
{
  #define PI 3.14
  typedef short CHAR;
  printf("sizeof CHAR in func: %d\n",sizeof(CHAR));
}

int main (void)
{
  printf("sizeof CHAR in main: %d\n",sizeof(CHAR));
  func();
  typedef int CHAR;
  printf("sizeof CHAR in main: %d\n",sizeof(CHAR));
  printf("PI:%f\n", PI);
  return 0;
}
```

宏定义在预处理阶段就已经替换完毕，是全局性的，只要保证引用它的地方在定义之后就可以了。而使用typedef声明的类型则跟普通变量一样遵循作用域规则。上面代码的运行结果为：

```
sizeof CHAR in main: 1
sizeof CHAR in func: 2
sizeof CHAR in main: 4
PI:3.140000
```

4、如何避免typedef的滥用?

通过上面的学习我们可以看到：使用typedef可以让我们的代码更加简洁、可读性更好。在实际的编程中，越来越多的人也开始尝试使用typedef，甚至到了“过犹不及”的滥用地步：但凡遇到结构体、联合、枚举都要用个typedef封装一下，不用就显得你low、你菜、你的代码没水平。

其实，typedef也有副作用，不一定非得处处都用它。比如上面我们封装的STUDENT类型，当你定义一个变量时：

```
STUDENT stu;
```

不看STUDENT的声明，你知道stu的含义吗？未必吧。而如果我们直接使用struct定义一个变量，则会更加清晰，让你一下子就知道stu是个结构体类型的变量：

```
struct  student stu;
```

一般来讲，当遇到以下情形时，使用typedef可能会有用，否则可能会适得其反：

- 创建一个新的数据类型
- 跨平台、指定长度的类型：如U32/U16/U8

- 跟操作系统、BSP、网络字宽相关的数据类型：如size_t、pid_t等
- 不透明的数据类型：需要隐藏结构体细节，只能通过函数接口访问的数据类型

在阅读Linux内核源码过程中，你会发现大量使用了typedef，哪怕是简单的int、long都使用了 typedef。这是因为Linux内核源码发展到今天，已经支持了太多的平台和CPU架构，为了保证数据的跨平台性和可移植性，所以很多时候不得已使用了typedef，对一些数据指定固定长度，如U8/U16/U32等。

但是，内核也不是到处到滥用，什么时候该用，什么不该用，也是有一定的规则要遵循的，具体大家可以看kernel Document中的CodingStyle中关于typedef的使用建议。

END

来源：果果小师弟

推荐阅读

为什么俄罗斯不担心芯片禁运?

偷偷盘点一下各大公司的实习薪资

高手常用的3个开源库，让单片机开发事半功倍!

→点关注，不迷路←

890篇原创内容

公众号

嵌入式ARM

关注这个时代最火的嵌入式ARM，你想知道的都在这里。

140篇原创内容

公众号

喜欢此内容的人还喜欢

顶级代码女神，编程界最有权势的女王

码农翻身

C语言：结构体就这样被攻克了！

玩转嵌入式

70行Go代码可以打败C！

硬件攻城狮