C语言中三块“难啃的硬骨头”

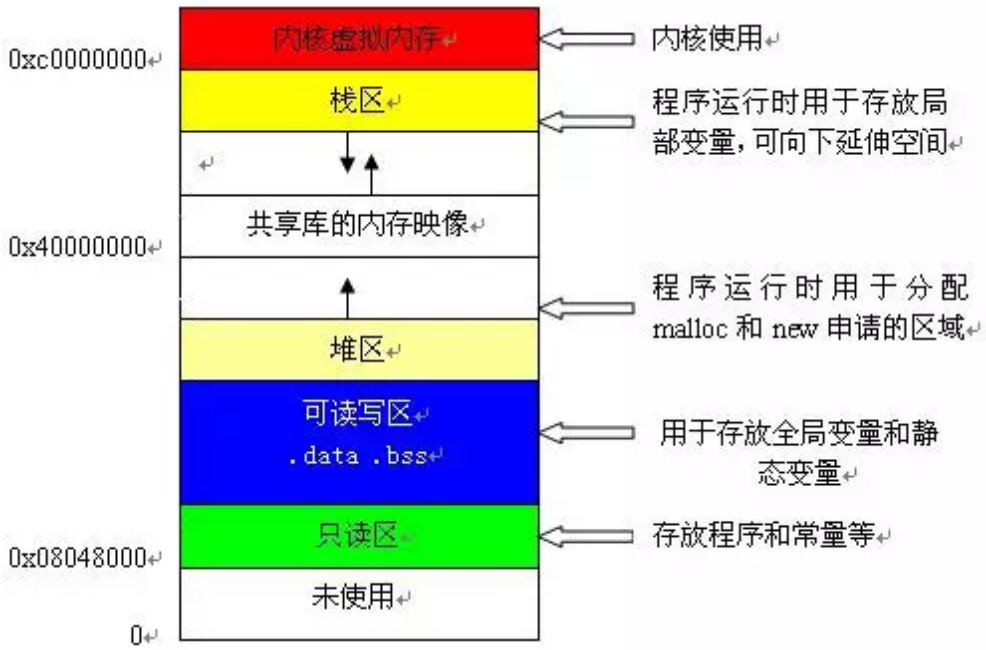
嵌入式ARM 2019-11-07

整理/付斌，参考《程序员互动联盟》

C语言在嵌入式学习中是必备的知识，审核大部分操作都要围绕C语言进行，而其中有三块“难啃的硬骨头”几乎是公认级别的。

01

指针



指针公认最难理解的概念，也是让很多初学者选择放弃的直接原因

指针之所以难理解，因为指针本身就是一个变量，是一个非常特殊的变量，专门存放地址的变量，这个地址需要给申请空间才能装东西，而且因为是个变量可以中间赋值，这么一倒腾很多人就开始犯晕了，绕不开弯了。C语言之所以被很多高手所喜欢，就是指针的魅力，中间可以灵活的切换，执行效率超高，这点也是让小白晕菜的地方。

指针是学习绕不过去的知识点，而且学完C语言，下一步紧接着切换到数据结构和算法，指针是切换的重点，指针搞不定下一步进行起来就很难，会让很多人放弃继续学习的勇气。

指针直接对接内存结构，常见的C语言里面的指针乱指，数组越界根本原因就是内存问题。在指针这个点有无穷无尽的发挥空间。很多编程的技巧都在此集结。

指针还涉及如何申请释放内存，如果释放不及时就会出现内存泄露的情况，指针是高效好用，但不彻底搞明白对于有些人来说简直就是噩梦。

在概念方面问题可以参见此前推文《对于C语言指针最详尽的讲解》，那么在指针方面可以参见一下大神的经验：

复杂类型说明

要了解指针，多多少少会出现一些比较复杂的类型。所以先介绍一下如何完全理解一个复杂类型。

要理解复杂类型其实很简单，一个类型里会出现很多运算符，他们也像普通的表达式一样，有优先级，其优先级和运算优先级一样。

所以笔者总结了一下其原则：从变量名处起，根据运算符优先级结合，一步一步分析。

下面让我们先从简单的类型开始慢慢分析吧。

- **int p;**

这是一个普通的整型变量

- **int p;**

首先从P处开始，先与结合，所以说明P是一个指针。然后再与int结合，说明指针所指向的内容的类型为int型，所以P是一个返回整型数据的指针

- **int p[3];**

首先从P处开始，先与[]结合，说明P是一个数组。然后与int结合，说明数组里的元素是整型的，所以P是一个由整型数据组成的数组。

- **int *p[3];**

首先从P处开始，先与[]结合，因为其优先级比高，所以P是一个数组。然后再与结合，说明数组里的元素是指针类型。之后再与int结合，说明指针所指向的内容的类型是整型的，所以P是一个由返回整型数据的指针所组成的数组。

- **int (*p)[3];**

首先从P处开始，先与结合，说明P是一个指针。然后再与[]结合(与"()"这步可以忽略，只是为了改变优先级)，说明指针所指向的内容是一个数组。之后再与int结合，说明数组里的元素是整型的。所以P是一个指向由整型数据组成3个整数的指针。

- **int **p;**

首先从P开始，先与*结合，说明P是一个指针。然后再与*结合，说明指针所指向的元素是指针。之后再与int结合，说明该指针所指向的元素是整型数据。由于二级指针以及更高级的指针极少用在复杂的类型中，所以后面更复杂的类型我们就不考虑多级指针了，最多只考虑一级指针。

- **int p(int);**

从P处起，先与()结合，说明P是一个函数。然后进入()里分析，说明该函数有一个整型变量的参数，之后再与外面的int结合，说明函数的返回值是一个整型数据。

- **Int (*p)(int);**

从P处开始，先与指针结合，说明P是一个指针。然后与()结合，说明指针指向的是一个函数。之后再与()里的int结合，说明函数有一个int型的参数，再与最外层的int结合，说明函数的返回类型是整型，所以P是一个指向有一个整型参数且返回类型为整型的函数的指针。

- **int (p(int))[3];**

可以先跳过，不看这个类型，过于复杂。从P开始，先与()结合，说明P是一个函数。然后进入()里面，与int结合，说明函数有一个整型变量参数。然后再与外面的结合，说明函数返回的是一个指针。之后到最外面一层，先与[]结合，说明返回的指针指向的是一个数组。接着再与结合，说明数组里的元素是指针，最后再与int结合，说明指针指向的内容是整型数据。所以P是一个参数为一个整数据且返回一个指向由整型指针变量组成的数组的指针变量的函数。

说到这里也就差不多了。理解了这几个类型，其它的类型对我们来说也是小菜了。不过一般不会用太复杂的类型，那样会大大减小程序的可读性，请慎用。这上面的几种类型已经足够我们用了。

细说指针

指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址。

要搞清一个指针需要搞清指针的四方面的内容：指针的类型、指针所指向的类型、指针的值或者叫指针所指向的内存区、指针本身所占据的内存区。让我们分别说明。

先声明几个指针放着做例子：

（1）int*ptr;
（2）char*ptr;
（3）int**ptr;
（4）int(*ptr)[3];
（5）int*(*ptr)[4];

指针的类型

从语法的角度看，小伙伴们只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。

让我们看看上述例子中各个指针的类型：

（1）intptr;//指针的类型是int
（2）charptr;//指针的类型是char
（3）intptr;//指针的类型是int
（4）int(ptr)[3];//指针的类型是int()[3]
（5）int*(ptr)[4];//指针的类型是int(*)[4]

怎么样？找出指针的类型的方法是不是很简单？

指针所指向的类型

当通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。

从语法上看，小伙伴们只需把指针声明语句中的指针名字和名字左边的指针声明符*去掉，剩下的就是指针所指向的类型。

上述例子中各个指针所指向的类型：

（1）intptr; //指针所指向的类型是int
（2）char*ptr; //指针所指向的的类型是char*
（3）int*ptr; //指针所指向的的类型是int*
（4）int(*ptr)[3]; //指针所指向的的类型是int(*)[3]
（5）int*(*ptr)[4]; //指针所指向的的类型是int*(*)[4]

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当小伙伴们对C 越来越熟悉时，就会发现，把与指针搅和在一起的"类型"这个概念分成"指针的类型"和"指针所指向的类型"两个概念，是精通指针的关键点之一。

笔者看了不少书，发现有些写得差的书中，就把指针的这两个概念搅在一起了，所以看起书来前后矛盾，越看越糊涂。

指针的值

即指针所指向的内存区或地址。

指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的

数值。

在32位程序里，所有类型的指针的值都是一个32位整数，因为32位程序里内存地址全都是32位长。指针所指向的内存区就是从指针的值所代表的那个内存地址开始，长度为si zeof(指针所指向的类型)的一片内存区。

以后，我们说一个指针的值是XX，就相当于说该指针指向了以XX为首地址的一片内存区域；我们说一个指针指向了某块内存区域，就相当于说该指针的值是这块内存区域的首地址。

指针所指向的内存区和指针所指向的类型是两个完全不同的概念。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

以后，每遇到一个指针，都应该问问：这个指针的类型是什么？指针指的类型是什么？该指针指向了哪里？

指针本身所占据的内存区

指针本身占了多大的内存？只要用函数sizeof(指针的类型)测一下就知道了。在32位平台里，指针本身占据4个字节的长度。指针本身占据的内存这个概念在判断一个指针表达式是否是左值时很有用。

02

函数概念

面向过程对象模块的基本单位，以及对应各种组合，函数指针，指针函数

一个函数就是一个业务逻辑块，是面向过程，单元模块的最小单元，而且在函数的执行过程中，形参，实参如何交换数据，如何将数据传递出去，如何设计一个合理的函数，

不单单是解决一个功能，还要看是不是能够复用，避免重复造轮子。

函数指针和指针函数，表面是两个字面意思的互换实际上含义截然不同，指针函数比较好理解，就是返回指针的一个函数，函数指针这个主要用在回调函数，很多人觉得函数都没还搞明白，回调函数更晕菜了。其实可以通俗的理解指向函数的指针，本身是一个指针变量，只不过在初始化的时候指向了函数，这又回到了指针层面。没搞明白指针再次深入的向前走特别难。

C语言的开发者们为后来的开发者做了一些省力气的事情，他们编写了大量代码，将常见的基本功能都完成了，可以让别人直接拿来使用。但是那么多代码，如何从中找到自己需要的呢？将所有代码都拿来显然是不太现实。

但是这些代码，早已被早期的开发者们分门别类地放在了不同的文件中，并且每一段代码都有唯一的名字。所以其实学习C语言并没有那么难，尤其是可以在动手锻炼做项目中

进行。使用代码时，只要在对应的名字后面加上( )就可以。这样的一段代码就是函数，函数能够独立地完成某个功能，一次编写完成后可以多次使用。

很多初学者可能都会把C语言中的函数和数学中的函数概念搞混淆。其实真相并没有那么复杂，C语言中的函数是有规律可循迹的，只要搞清楚了概念你会发现还挺有意思的。函数的英文名称是 Function，对应翻译过来的中文还有“功能”的意思。C语言中的函数也跟功能有着密切的关系。

我们来看一小段C语言代码：

```
1 #include<stdio.h>
2 int main()
3 {
4 puts("Hello World");
5 return 0;
6 }
```

把目光放在第4行代码上，这行代码会在显示器上输出“Hello World”。前面我们已经讲过，puts 后面要带( )，字符串也要放在( )中。

在C语言中，有的语句使用时不能带括号，有的语句必须带括号。带括号的就是函数（Function）。

C语言提供了很多功能，我们只需要一句简单的代码就能够使用。但是这些功能的底层都比较复杂，通常是软件和硬件的结合，还要要考虑很多细节和边界，如果将这些功能都交给程序员去完成，那将极大增加程序员的学习成本，降低编程效率。

有了函数之后，C语言的编程效率就好像有了神器一样，开发者们只需要随时调用就可以

了，像进程函数、操作函数、时间日期函数等都可以帮助我们直接实现C语言本身的功能。

C语言函数是可以重复使用的。

函数的一个明显特征就是使用时必须带括号( )，必要的话，括号中还可以包含待处理的数据。例如puts("尚观科技")就使用了一段具有输出功能的代码，这段代码的名字是puts，"尚观科技" 是要交给这段代码处理的数据。使用函数在编程中有专业的称呼，叫做函数调用（Function Call）。

如果函数需要处理多个数据，那么它们之间使用逗号,分隔，例如：
pow(10, 2);
该函数用来求10的2次方。

好了，看到这里你有没有觉得其实C语言函数还是比较有意思的，而且并没有那么复杂困难。以后再遇到菜鸟小白的时候，你一口一个C语言的函数，说不定就能当场引来无数膜拜的目光。

03

结构体，递归

很多在大学学习C语言的，很多课程都没学完，结构体都没学到，因为从章节的安排来看好像，结构体学习放在教材的后半部分了，弄得很多学生觉得结构体不重要，如果只是应付学校的考试，或者就是为了混个毕业证，的确学的意义不大。

如果想从事编程这个行业，对这个概念还不了解，基本上无法构造数据模型，没有一个业务体是完全使用原生数据类型来完成的，很多高手在设计数据模型的时候，一般先把头文件中的结构体数据整理出来。然后设计好功能函数的参数，以及名字，然后才真正开始写c源码。

如果从节省空间考虑结构体里面的数据放的顺序不一样在内存中占用的空间也不一样，结构体与结构体之间赋值，结构体存在指针那么赋值要特别注意，需要进行深度的赋值。

递归一般用于从头到位统计或者罗列一些数据，在使用的时候很多初学者都觉得别扭，怎么还能自己调用自己？而且在使用的时候，一定设置好跳出的条件，不然无休止的进行下去，真就成无线死循环了。

对于结构体方面的知识，可以参见此前推送的文章《C语言结构体（struct）最全的讲解（万字干货）》。具体也可以参见大佬的经验：

相信大家对于结构体都不陌生。在此，分享出本人对C语言结构体的研究和学习的总结。如果你发现这个总结中有你以前所未掌握的，那本文也算是有点价值了。当然，水平有限，若发现不足之处恳请指出。代码文件test.c我放在下面。
在此，我会围绕以下2个问题来分析和应用C语言结构体：

1. C语言中的结构体有何作用
2. 结构体成员变量内存对齐有何讲究**(**重点**)**

对于一些概念的说明，我就不把C语言教材上的定义搬上来。我们坐下来慢慢聊吧。

1. 结构体有何作用

三个月前，教研室里一个学长在华为南京研究院的面试中就遇到这个问题。当然，这只是面试中最基础的问题。如果问你你怎么回答?
我的理解是这样的，C语言中结构体至少有以下三个作用：

(1) 有机地组织了对象的属性。

比如，在STM32的RTC开发中，我们需要数据来表示日期和时间，这些数据通常是年、月、日、时、分、秒。如果我们不用结构体，那么就需要定义6个变量来表示。这样的话程序的数据结构是松散的，我们的数据结构最好是“高内聚，低耦合”的。所以，用一个结构体来表示更好，无论是从程序的可读性还是可移植性还是可维护性皆是：

```
typedef struct //公历日期和时间结构体
{
vu16 year;
vu8 month;
vu8 date;
vu8 hour;
vu8 min;
vu8 sec;
}_calendar_obj;
_calendar_obj calendar; //定义结构体变量
```

(2) 以修改结构体成员变量的方法代替了函数**(**入口参数**)**的重新定义。

如果说结构体有机地组织了对象的属性表示结构体“中看”，那么以修改结构体成员变量的方法代替函数(入口参数)的重新定义就表示了结构体“中用”。继续以上面的结构体为例子，我们来分析。假如现在我有如下函数来显示日期和时间：

```
void DsipDateTime( _calendar_obj DateTimeVal)
```

那么我们只要将一个_calendar_obj这个结构体类型的变量作为实参调用

DsipDateTime()即可，DsipDateTime()通过DateTimeVal的成变量来实现内容的显示。如果不用结构体，我们很可能需要写这样的一个函数：

```
1  void DsipDateTime( vu16 year, vu8 month, vu8 date, vu8 hour, vu8 min,
```

显然这样的形参很不可观，数据结构管理起来也很繁琐。如果某个函数的返回值得是一个表示日期和时间的数据，那就更复杂了。这只是一方面。
另一方面，如果用户需要表示日期和时间的数据中还要包含星期(周)，这个时候，如果之前没有用机构体，那么应该在DsipDateTime()函数中在增加一个形参vu8 week：

```
1  void DsipDateTime( vu16 year, vu8 month, vu8 date, vu8 week, vu8 hour
```

可见这种方法来传递参数非常繁琐。所以以结构体作为函数的入口参数的好处之一就是函数的声明void DsipDateTime(_calendar_obj DateTimeVal)不需要改变，只需要增加结构体的成员变量，然后在函数的内部实现上对calendar.week作相应的处理即可。这样，在程序的修改、维护方面作用显著。

```
1  typedef struct //公历日期和时间结构体
2  {
3  vu16 year;
4  vu8 month;
5  vu8 date;
6  vu8 week;
7  vu8 hour;
8  vu8 min;
9  vu8 sec;
10 }_calendar_obj;
```

```
_calendar_obj calendar; //定义结构体变量
```

(3) 结构体的内存对齐原则可以提高**CPU**对内存的访问速度**(**以空间换取时间**)**。

并且，结构体成员变量的地址可以根据基地址(以偏移量offset)计算。我们先来看看下面的一段简单的程序，对于此程序的分析会在第2部分结构体成员变量内存对齐中详细说明。

```
#include<stdio.h>

int main()
{
    struct    //声明结构体char_short_long
    {
        char  c;
        short s;
        long  l;
    }char_short_long;

    struct    //声明结构体long_short_char
    {
        long  l;
        short s;
        char  c;
    }long_short_char;

    struct    //声明结构体char_long_short
    {
        char  c;
        long  l;
        short s;
```

```
    }char_long_short;

printf(" \n");
printf(" Size of char   = %d bytes\n",sizeof(char));
printf(" Size of shrot  = %d bytes\n",sizeof(short));
printf(" Size of long   = %d bytes\n",sizeof(long));
printf(" \n");  //char_short_long
printf(" Size of char_short_long       = %d bytes\n",sizeof(char_s
printf("     Addr of char_short_long.c = 0x%p (10进制：%d)\n",&char_
printf("     Addr of char_short_long.s = 0x%p (10进制：%d)\n",&char_
printf("     Addr of char_short_long.l = 0x%p (10进制：%d)\n",&char_
printf(" \n");

printf(" \n");  //long_short_char
printf(" Size of long_short_char       = %d bytes\n",sizeof(long_s
printf("     Addr of long_short_char.l = 0x%p (10进制：%d)\n",&long_
printf("     Addr of long_short_char.s = 0x%p (10进制：%d)\n",&long_
printf("     Addr of long_short_char.c = 0x%p (10进制：%d)\n",&long_
printf(" \n");

printf(" \n");  //char_long_short
printf(" Size of char_long_short       = %d bytes\n",sizeof(char_l
printf("     Addr of char_long_short.c = 0x%p (10进制：%d)\n",&char_
printf("     Addr of char_long_short.l = 0x%p (10进制：%d)\n",&char_
printf("     Addr of char_long_short.s = 0x%p (10进制：%d)\n",&char_
printf(" \n");
return 0;
}
```

程序的运行结果如下(注意：括号内的数据是成员变量的地址的十进制形式)：

2. 结构体成员变量内存对齐

首先，我们来分析一下上面程序的运行结果。前三行说明在我的程序中，char型占1个字节，short型占2个字节，long型占4个字节。char_short_long、long_short_char和char_long_short是三个结构体成员相同但是成员变量的排列顺序不同。并且从程序的运行结果来看，

```
Size of char_short_long = 8 bytes
Size of long_short_char = 8 bytes
Size of char_long_short = 12 bytes //比前两种情况大4 byte ！
```

并且，还要注意到，1 byte (char)+ 2 byte (short)+ 4 byte (long) = 7 byte，而不是8 byte。

所以，结构体成员变量的放置顺序影响着结构体所占的内存空间的大小。一个结构体变量所占内存的大小不一定等于其成员变量所占空间之和。如果一个用户程序或者操作系统(比如uC/OS-II)中存在大量结构体变量时，这种内存占用必须要进行优化，也就是说，结构体内部成员变量的排列次序是有讲究的。

结构体成员变量到底是如何存放的呢?

在这里，我就不卖关子了，直接给出如下结论，在没有#pragma pack宏的情况下：

原则**1**　结构（struct或联合union）的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员存储的起始位置要从该成员大小的整数倍开始（比如int在32位机为4字节，则要从4的整数倍地址开始存储）。

原则**2**　结构体的总大小，也就是sizeof的结果，必须是其内部最大成员的整数倍，不足的要补齐。

*原则**3**　结构体作为成员时，结构体成员要从其内部最大元素大小的整数倍地址开始存储。（struct a里存有struct b，b里有char，int，double等元素时，那么b应该从8的整数倍地址处开始存储，因为sizeof(double) = 8 bytes）

这里，我们结合上面的程序来分析(暂时不讨论原则3)。

先看看char_short_long和long_short_char这两个结构体，从它们的成员变量的地址可以看出来，这两个结构体符合原则1和原则2。注意，在 char_short_long的成员变量的地址中，char_short_long.s的地址是1244994，也就是说，1244993是“空的”，只是被“占位”了！

再看看char_long_short这个结构体，char_long_short的地址分布情况如下表：

成员变量	成员变量十六进制地址	成员变量十进制地址
char_long_short.c	0x0012FF2C	1244972

char_long_short.l	0x0012FF30	1244976
char_long_short.s	0x0012FF34	1244980

可见，其内存分布图如下，共12 bytes：

地址	1244972	1244973	1244974	1244975	1244976	1244977	1244978	1244979	1244980	1244981	1244982	1244983
成员	.c				.l				.s			

首先，1244972能被1整除，所以char_long_short.c放在1244972处没有问题(其实，就char型成员变量自身来说，其放在任何地址单元处都没有问题)，根据原则1，在之后的1244973~1244975中都没有能被4(因为sizeof(long)=4bytes)整除的，1244976能被4整除，所以char_long_short.l应该放在1244976处，那么同理，最后一个.s(sizeof(short)=2 bytes)是应该放在1244980处。

是不是这样就结束了？不是，还有原则2。根据原则2的要求，char_long_short这个结构体所占的空间大小应该是其占内存空间最大的成员变量的大小的整数倍。如果我们到此就结束了，那么char_long_short所占的内存空间是1244972~1244981共计10bytes，不符合原则2，所以，必须在最后补齐2个 bytes(1244982~1244983)。

至此，一个结构体的内存布局完成了。

下面我们按照上述原则，来验证这样的分析是不是正确。按上面的分析，地址单元1244973、1244974、1244975以及1244982、1244983都是空的(至少char_long_short未用到，只是“占位”了)。如果我们的分析是正确的，那么，定义这样一个结构体，其所占内存也应该是12 bytes：

```
struct //声明结构体char_long_short_new
{
char c;
char add1; //补齐空间
char add2; //补齐空间
char add3; //补齐空间
long l;
short s;
char add4; //补齐空间
char add5; //补齐空间
}char_long_short_new;
```

运行结果如下：

可见，我们的分析是正确的。至于原则3，大家可以自己编程验证，这里就不再讨论了。

所以，无论你是在VC6.0还是Keil C51，还是Keil MDK中，当你需要定义一个结构体时，只要你稍微留心结构体成员变量内存对齐这一现象，就可以在很大程度上节约MCU的RAM。这一点不仅仅应用于实际编程，在很多大型公司，比如IBM、微软、百度、华为的笔试和面试中，也是常见的。

这三大块硬骨头是学习C语言的绊脚石，下功夫拿掉基本上C语言的大动脉就打通了，那么再去学习别的内容就相对比较简单了。编程学习过程中越是痛苦的时候，学到的东西就会越多，克服过去就会自己的技能，放弃了前面的付出的时间都将清零。越是难学的语言在入门之后，在入门之后越觉得过瘾，而且还容易上瘾。你上瘾了没？还是放弃

了?

-END-

推荐阅读

【01】深度：震惊世间的惊人代码（附完整代码）
【02】编译器如何将高级语言转化成汇编语言的?
【03】C语言在嵌入式系统编程时的注意事项
【04】由C语言编写的C编译器是怎样来的?
【05】还没搞懂C语言指针？最详细的干货讲解
【06】C语言结构体（struct）最全的讲解
【07】为什么在C语言中，goto这么不受待见?

Modified on 2019-11-07

喜欢此内容的人还喜欢

从零造单片机需要哪些知识?

嵌入式ARM

男女交往指南：不贪，不缠，主动

网易公开课

中国城市，正面临的四大生死之战！

大佬动向