**C语言关键字auto与register的深入理解**



[Bee大前端](https://blog.csdn.net/u011625768) 2015-07-02 10:26:11  5290  收藏 31

分类专栏： [c 语言](https://blog.csdn.net/u011625768/category_5620799.html) 文章标签： [c语言](https://www.csdn.net/tags/MtTaEg0sMzc0NzktYmxvZwO0O0OO0O0O.html) [编译器](https://www.csdn.net/tags/MtTaEg0sMTA0NzctYmxvZwO0O0OO0O0O.html) [32位](https://so.csdn.net/so/search/s.do?q=32%E4%BD%8D&t=blog&o=vip&s=&l=&f=&viparticle=) [计算机](https://so.csdn.net/so/search/s.do?q=%E8%AE%A1%E7%AE%97%E6%9C%BA&t=blog&o=vip&s=&l=&f=&viparticle=) [语言](https://www.csdn.net/tags/MtTaEg0sMDU5NTUtYmxvZwO0O0OO0O0O.html)

版权

关键字，又叫保留字，是编译器能识别的特殊单词，每种计算机语言都会有其特定的关键字，C语言中有32位**关键字。  
问：为什么要有关键字？  
答：关键字是程序设计中代码必须包含的部分，编译器在编译C代码的时候，必然要将C代码进行断句，将代码分割成不同部分，将这些部分分别进行解析和编译。**  
int a = 10;  int是关键字，编译器看到它出现，会将它后面的字符作为整型变量名来处理。  
也就是说，关键字是编译器能认识的特殊字符串符号。  
关键字的数量是由编译器来决定的，关键字大小写敏感性也和编译器有关。如果关键字写错，那么在代码的解析过程中，编译器就会报错：符号不能识别或符号不能被解析。  
每个关键字有着不同的意义，用来告知编译器编程者的目的。  
  
**关键字分类**  
32个关键字每个都有不同的意义，大体上根据其意义可以分为以下几类（下划线表示不同分类中有交集）：  
**1)非常见：**auto、register、volatile、goto  
**2)存储相关：**const、extern、register、volatile、static、auto、signed、unsigned  
**3)数据类型：**char、short、int、float、long、double、struct、union、enum、void  
**4)逻辑控制：**if、else、for、while、do、break、continue、return、default、switch、case、goto  
**5)特殊用途：**sizeof、typedef  
我相信，大部分关键字我们都能认识，并且能够使用，有一部分可能很少见，甚至一点印象也没有：它也是C**语言的关键字？？？  
1.隐形刺客：auto  
描述：**auto关键字在我们写的代码里几乎看不到，但是它又无处不在，它是如此的重要，又是如此的与世无争，默默的履行着自己的义务，却又隐姓埋名。  
**作用：**C程序是面向过程的，在C代码中会出现大量的函数模块，每个函数都有其生命周期（也称作用域），在函数生命周期中声明的变量通常叫做局部变量，也叫自动变量。例如：

复制代码代码如下:

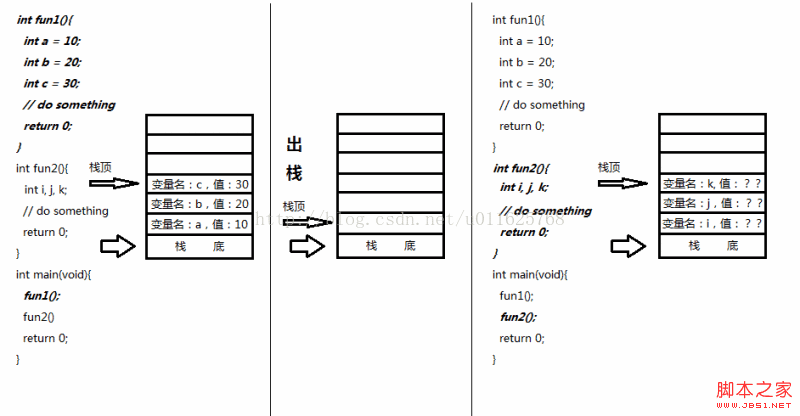
    int fun(){    
          int a = 10;      // auto int a = 10;    
          // do something    
          return 0;    
    }

复制代码代码如下:

    int fun(){    
          int a = 10;      // auto int a = 10;    
          // do something    
          return 0;    
    }

整型变量a在fun函数内声明，其作用域为fun函数内，出来fun函数，不能被引用，a变量为自动变量。也就是说编译器会有int a = 10之前会加上auto的关键字。  
auto的出现意味着，当前变量的作用域为当前函数或代码段的局部变量，意味着当前变量会在内存栈上进行分配。  
内存栈：  
如果大家学过数据结构，应该知道，栈就是先进后出的数据结构。它类似于我们用箱子打包书本，第一本扔进去大英，第二本扔进行高数，第三本扔进行小说，那么取书的时候，先取出来第一本是小说，第二是高数，第三本是大英。  
栈的操作为入栈和出栈，入栈就是向箱子里扔书，出栈就是从箱子里取书。那么这和我们的auto变量分配空间有什么关系呢？

由于一个程序中可能会有大量的变量声明，每个变量都会占有一定的内存空间，而内存空间对于计算机来说是宝贵的硬件资源，因此合理的利用内存是编译器要做的一个主要任务。有的变量是一次性使用的，如局部变量。有的变量要伴随着整个程序来使用的，如全局变量。为了节省内存空间，优化性能，编译器通常会将一次性使用的变量分配在栈上。也就是说，代码中声明一个一次性变量，就在栈上进行入栈操作。当该变量使用完了（生命周期结束），进行出栈操作。这样，在执行不同的函数的时候，就会在一个栈上进行出入栈操作，也就是说它们在频繁的使用一个相同的内存空间，从而可以更高效的利用内存。



**PS：**有的编译器为了提高效率，在出栈时不会进行数据清空，这也就意味着，下个函数里的变量在入栈使用该空间时，里面的数据是上一次变量操作的结果。  
**2.闪电飞刀：register**  
描述：register就和它的名字一样，很少出现在代码世界中，因为敢称为闪电飞刀的变量，通常只会在一些特定场合才能出现。它是如此的快，以致于CPU都对其刮目相看，但是它有一个致命的缺点，它的速度“看心情”而定，不是每一次都能让人满意。  
作用：如果一个变量被register来修辞，就意味着，该变量会作为一个寄存器变量，让该变量的访问速度达到最快。比如：一个程序逻辑中有一个很大的循环，循环中有几个变量要频繁进行操作，这些变量可以声明为register类型。  
**寄存器变量：**寄存器变量是指一个变量直接引用寄存器，也就是对变量名的操作的结果是直接对寄存器进行访问。寄存器是CPU的亲信，CPU操作的每个操作数和操作结果，都由寄存器来暂时保存，最后才写入到内存或从内存中读出。也就是说，变量的值通常保存在内存中，CPU对变量进行读取先是将变量的值从内存中读取到寄存器中，然后进行运算，运算完将结果写回到内存中。为什么要这么设计，而不直接对变量的值从内存中进行运算，而要再借助于寄存器？这是由于考虑到性能的问题才这么设计的。在计算机系统中，包含有很多种不同类型的存储器，如表xxx所示。  
**表xxx 计算机存储器分类**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 速度 | 特点 | 用途 |
| 静态存储器 | 最快 | 造价高，体积大，适合小容量的缓存 | 寄存器，缓存 |
| 动态存储器 | 较快 | 造价较低，体积较小，适合大容易保存数据 | 内存 |

在计算机中CPU的运算速度最快，现在都达到3GHZ左右，而相对应的存储器速度却相对慢很多，访问速度最快的寄存器和缓存，由于其体积又大，不适合大容量的使用，所以只能二者相接合的方式来提高效率。程序代码保存在内存中，当使用数据时，将其送到寄存器，让CPU来访问，使用完毕，送回内存保存。而C语言又允许使用寄存器来保存变量的值，很明显这样能大大提高程序的执行速度，但是，寄存器的个数是有限的，X86也就是十几个，ARM最多才37个。我们不可能将全部的变量都声明为寄存器变量，因为其它代码也要使用寄存器，同样，我们声明的寄存器变量也不一定直接保存在寄存器中，因为寄存器可能全部都在被其它代码占用。编译器只能是尽量的为我们的变量安排在寄存器中。  
在使用寄存器变量时，请注意：  
待声明为寄存器变量类型应该是CPU寄存器所能接受的类型，意味着寄存器变量是单个变量，变量长度应该小于等于寄存器长度  
不能对寄存器变量使用取地址符“&”，因为该变量没有内存地址  
尽量在大量频繁的操作时使用寄存器变量，且声明的变量个数应该尽量的少

  register：这个关键字请求编译器尽可能的将变量存在CPU内部寄存器中，而不是通过内存寻址访问，以提高效率。**注意是尽可能，不是绝对**。你想想，一个CPU 的寄存器也就那么几个或几十个，你要是定义了很多很多register 变量，它累死也可能不能全部把这些变量放入寄存器吧，轮也可能轮不到你。

一、皇帝身边的小太监----寄存器

    不知道什么是寄存器？那见过太监没有？没有？其实我也没有。没见过不要紧，见过就麻烦大了。^\_^，大家都看过古装戏，那些皇帝们要阅读奏章的时候，大臣总是先将奏章交给皇帝旁边的小太监，小太监呢再交给皇帝同志处理。这个小太监只是个中转站，并无别的功能。  
  
    好，那我们再联想到我们的CPU。CPU 不就是我们的皇帝同志么？大臣就相当于我们的内存，数据从他这拿出来。那小太监就是我们的寄存器了（这里先不考虑CPU 的高速缓存区）。数据从内存里拿出来先放到寄存器，然后CPU 再从寄存器里读取数据来处理，处理完后同样把数据通过寄存器存放到内存里，CPU 不直接和内存打交道。这里要说明的一点是:小太监是主动的从大臣手里接过奏章，然后主动的交给皇帝同志，但寄存器没这么自觉，它从不主动干什么事。一个皇帝可能有好些小太监，那么一个CPU 也可以有很多寄存器，不同型号的CPU 拥有寄存器的数量不一样。  
  
    为啥要这么麻烦啊？速度！就是因为速度。寄存器其实就是一块一块小的存储空间，只不过其存取速度要比内存快得多。进水楼台先得月嘛，它离CPU 很近，CPU 一伸手就拿到数据了，比在那么大的一块内存里去寻找某个地址上的数据是不是快多了？那有人问既然它速度那么快，那我们的内存硬盘都改成寄存器得了呗。我要说的是：你真有钱！

**二、举例**

    register修饰符暗示编译程序相应的变量将被频繁地使用，如果可能的话，应将其保存在CPU的寄存器中，以加快其存储速度。例如下面的内存块拷贝代码，

　　#ifdef NOSTRUCTASSIGN

　　memcpy (d, s, l)

　　{

        register char \*d;

　　    register char \*s;

　　    register int i;

　　    while (i--)

　　        \*d++ = \*s++;

　　}

　　#endif

**三、使用register 修饰符的注意点**

    但是使用register修饰符有几点限制。

　　首先，register**变量必须是能被CPU所接受的类型**。这通常意味着register变量**必须是一个单个的值**，并且**长度应该小于或者等于整型的长度**。不过，有些机器的寄存器也能存放浮点数。

　　其次，因为register变量可能不存放在内存中，所以不能用“&”来获取register变量的地址。

　　由于寄存器的数量有限，而且某些寄存器只能接受特定类型的数据（如指针和浮点数），因此真正起作用的register修饰符的数目和类型都依赖于运行程序的机器，而任何多余的register修饰符都将被编译程序所忽略。

　　在某些情况下，把变量保存在寄存器中反而会降低程序的运行速度。因为被占用的寄存器不能再用于其它目的；或者变量被使用的次数不够多，不足以装入和存储变量所带来的额外开销。

　　早期的C编译程序不会把变量保存在寄存器中，除非你命令它这样做，这时register修饰符是C语言的一种很有价值的补充。然而，随着编译程序设计技术的进步，在决定那些变量应该被存到寄存器中时，现在的C编译环境能比程序员做出更好的决定。实际上，许多编译程序都会忽略register修饰符，因为尽管它完全合法，但它仅仅**是暗示而不是命令**。

更多内容，欢迎同步关注作者公众号二维码！  
如果对你有用，麻烦关注一下，您的肯定是最大的支持！