**1.volatile**

　　特点：告诉编译器不要对变量的访问进行优化，每次都只从内存中取；对于有并行如状态寄存器、有中断服务程序或者多线程共享的变量，应使用此修饰符。

volatile是易变的，不稳定的意思，volatile是关键字，是一种类型修饰符，用它修饰的变量表示可以被某些编译器未知的因素更改，比如操作系统、硬件或者其他线程等，遇到这个关键字声明的变量，编译器对访问该变量的代码不在进行优化，从而可以提供对特殊地址的稳定访问。那么什么是编译器优化呢？

　　为了提高运行效率，攻城湿们可是费尽心机地把代码优化，把程序运行时存取速度优化。一般，分为**硬件优化**和**软件优化**。硬件优化，流水线工作，详细可以参考《计算机组成原理》。软件优化，一部分是程序猿们做的**代码优化**（前提你得有优化的思路和能力），还有一部分就是我们的**编译器优化**了。现代的编译器经过那么多年的发展，已经比较成熟，把很多余的变量忽略掉，让代码的效率更高。而在默认情况下，编译器都会对代码进行优化，为了提高存取变量的速度，会把一些变量在**寄存器**（要知道控制器、运算器和寄存器是CPU的三大组成成分）里存取，而不是在内存里存取了，那这样CPU在自己家里拿东西当然比从内存那里拿东西要快得多。举个小栗子：

　　　　int i = 5;  
　　　　int a = i;  
　　　　……  
　　　　int b = i;  
　　编译器发现两次从i读数据的代码之间，并没有对i进行过操作，它会自动把上次读的数据放在b中，而不是重新从i里面读取。

　　而volatile 关键字告诉编译器该变量是随时可能发生变化的，每次使用它的时候必须从**内存**中取出他的值，因而编译器生成的汇编代码会从原内存地址中读取数据使用，而不是从寄存器或者缓存中读取，从而保证了对特殊地址的稳定访问。也就是说，状态要经常变化的，为了防止我们编译优化而导致的存取的数据不同步的问题，这时我们就需要用到！那具体到什么场景下需要用到volatile关键字呢？

　　1、并行设备的硬件寄存器（如：状态寄存器）；

　　2、一个中断服务子程序中会访问到的**非自动变量**(Non-automatic variables)；

　　3、多线程应用中被几个任务共享的变量

　　上面提到了非自动变量，这里就进一步对几种变量做一番解释：

**自动变量**：是在函数内部定义和使用的变量，它是**局部变量**。

**非自动变量**：有两种，一种是**全局变量**，一种是**静态变量**。

　　全局变量：在函数外面定义的变量，只能定义一次，不能有重复的定义，不然就会发生错误，而其他的文件要想使用这个变量，需要extern来声明这个变量（也可省略，因为默认就是extern），这个声明叫做引用声明。若不想被其他文件访问，则用static关键字声明为静态变量。静态变量与自动变量的本质区别是，静态变量并不像自动变量那样使用堆栈机制来使用内存。而是为静态变量分配固定的内存，在程序运行的整个过程中，它都会被保持，而不会不销毁。这就是说静态变量的持续性是程序运行的整个周期。这有利于我们共享一些数据。如果静态变量在函数内部定义，则它的作用域就是在这个函数内部，仅在这个函数内部使用它才有效，但是它不同于自动变量的，自动变量离开函数后就会别销毁，而静态变量不会被销毁。他在函数的整个运行周期内都会存在。

**1.register**变

特点：存在CPU寄存器，访问可能提速；不能被取地址。

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。

因为，如果定义了很多register变量，可能会超过CPU的寄存器个数，超过容量。所以只是可能。

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。

　　2.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。

3.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。

标准输入输出语句  
Turbo C 2.0标准库提供了两个控制台格式化输入、输出函数scanf();和printf();这两个函数可以在标准输入输出设备上以各种不同的格式读写数据。scanf() 函数用来从标准输入设备(键盘)上读数据，printf()函数用来向标准输出设备(屏幕)写数据。下面详细介绍这两个函数的用法。  
1.标准输入语句  
scanf()函数是格式化输入函数, 它从标准输入设备(键盘) 读取输入的信息。其调用格式为:   
scanf(<格式化字符串>, <地址表>);  
格式化字符串包括以下三类不同的字符;   
(1).空白字符:空白字符会使scanf()函数在读操作中略去输入中的一个或多个空白字符。  
(2).非空白字符:一个非空白字符会使scanf()函数在读入时剔除掉与这个非空白字符相同的字符。  
(3).格式化说明符:以%开始,后跟一个或几个规定字符,用来确定输出内容格式。  
Turbo C 2.0提供的输入格式化规定符如下:   
━━━━━━━━━━━━━━━━━━━━━━━━━━   
符号 作用   
──────────────────────────   
%d 十进制有符号整数   
%u 十进制无符号整数   
%f 浮点数   
%s 字符串   
%c 单个字符   
%p 指针的值   
%x,%X 无符号以十六进制表示的整数   
%o 无符号以八进制表示的整数   
━━━━━━━━━━━━━━━━━━━━━━━━━━  
地址表是需要读入的所有变量的地址,而不是变量本身，取地址符为'&'。各个变量的地址之间同,分开。  
例如：  
scanf(%d,%d,&i,&j);   
上例中的scanf()函数先读一个整型数,然后把接着输入的逗号剔除掉,最后读入另一个整型数。如果,这一特定字符没有找到,scanf()函数就终止。若参数之间的分隔符为空格,则参数之间必须输入一个或多个空格。  
说明:   
(a).对于各个变量，类型说明符是什么，输入格式化说明符就应该用对应的类型。否则会出现程序错误或输入数据和理想的不一样。  
(b).对于字符串数组或字符串指针变量,由于数组名和指针变量名本身就是地址,因此使用scanf()函数时,不需要在它们前面加上&操作符。   
char \*p,str[20];   
scanf(%s, p);  
scanf(%s, str);   
具体字符串，指针的知识以后再介绍。  
(c).可以在格式化字符串中的%各格式化规定符之间加入一个整数,表示任何读操作中的最大位数。   
如上例中若规定只能输入10字符给字符串指针p,则第一条scanf()函数语句变为:  
scanf(%10s, p);  
程序运行时一旦输入字符个数大于10, p就不再继续读入。  
实际使用scanf()函数时存在一个问题, 下面举例进行说明:   
当使用多个scanf()函数连续给多个字符变量输入时, 例如:   
char c1, c2;   
scanf(%c, &c1);   
scanf(%c, &c2);   
运行该程序,输入一个字符A后回车(要完成输入必须回车),在执行scanf(%c,&c1)时,给变量c1赋值A,但回车符仍然留在缓冲区内,执行输入语句scanf(%c,&c2)时,变量c2输出的是一空行,如果输入AB后回车,那么实际存入变量里的结果为c1为A，c2为B。  
要解决以上问题, 可以在输入函数前加入清除函数fflush();(这个函数的使用方法将在本节最后讲述)。  
(d).当在格式说明符之间加入'\*'时，表示跳过输入，例如：  
scanf(%3\*d,&a);  
当输入12345的时候，前面三个字符跳过去不考虑，最终变量a的值为45。  
2.标准输出语句  
printf()函数是格式化输出函数,一般用于向标准输出设备按规定格式输出信息。在编写程序时经常会用到此函数。printf()函数的调用格式为:   
printf(<格式化字符串>, <参量表>);   
其中格式化字符串包括两部分内容:一部分是正常字符,   
这些字符将按原样输出;另一部分是格式化规定字符,以%开始,后跟一个或几个规定字符,用来确定输出内容格式。   
参量表是需要输出的一系列参数,其个数必须与格式化字符串所说明的输出参数个数一样多,各参数之间用,分开,且顺序一一对应,否则将会出现意想不到的错误。  
对于输出语句，还有两个格式化说明符  
符号 作用  
%e 指数形式的浮点数  
%g 自动选择合适的表示法  
说明:   
(1).可以在%和字母之间插进数字表示最大场宽。   
例如: %3d 表示输出3位整型数,不够3位右对齐。   
%9.2f 表示输出场宽为9的浮点数,其中小数位为2,整数位为6,小数点占一位,不够9位右对齐。   
%8s 表示输出8个字符的字符串,不够8个字符右对齐。   
如果字符串的长度、或整型数位数超过说明的场宽,将按其实际长度输出。但对浮点数,若整数部分位数超过了说明的整数位宽度,将按实际整数位输出;若小数部分位数超过了说明的小数位宽度,则按说明的宽度以四舍五入输出。  
另外,若想在输出值前加一些0, 就应在场宽项前加个0。   
例如: %04d 表示在输出一个小于4位的数值时,将在前面补0使其总宽度为4位。   
如果用浮点数表示字符或整型量的输出格式,小数点后的数字代表最大宽度,小数点前的数字代表最小宽度。   
例如: %6.9s 表示显示一个长度不小于6且不大于9的字符串。若大于9,则第9个字符以后的内容将被删除。   
(2). 可以在%和字母之间加小写字母l,表示输出的是长型数。  
例如: %ld 表示输出long整数  
%lf 表示输出double浮点数  
(3). 可以控制输出左对齐或右对齐，即在%和字母之间加入一个- 号可说明输出为左对齐,否则为右对齐。   
例如: %-7d 表示输出7位整数左对齐   
%-10s 表示输出10个字符左对齐   
一些特殊规定字符(可以参照前面说的转义字符)  
━━━━━━━━━━━━━━━━━━━━━━━━━━  
字符 作用  
──────────────────────────  
\n 换行  
\f 清屏并换页  
\r 回车  
\t Tab符  
\xhh 表示一个ASCII码用16进表示  
━━━━━━━━━━━━━━━━━━━━━━━━━━  
由本节所学的printf()函数, 并结合上一节学习的数据类型, 看下面的语句,加深对Turbo C 2.0数据类型的了解。  
  
char c;   
int a=1234;   
float f=3.141592653589;   
double x=0.12345678987654321;   
c='\x41';   
printf(a=%d\n, a); /\*结果输出十进制整数a=1234\*/  
printf(a=%6d\n, a); /\*结果输出6位十进制数a= 1234\*/  
printf(a=%06d\n, a); /\*结果输出6位十进制数a=001234\*/  
printf(a=%2d\n, a); /\*a超过2位, 按实际值输出a=1234\*/  
printf(f=%f\n, f); /\*输出浮点数f=3.141593\*/  
printf(f=6.4f\n, f); /\*输出6位其中小数点后4位的浮点数f=3.1416\*/   
printf(x=%lf\n, x); /\*输出长浮点数x=0.123457\*/  
printf(x=%18.16lf\n,x); /\*输出18位其中小数点后16位的长浮点数x=0.1234567898765432\*/  
printf(c=%c\n, c); /\*输出字符c=A\*/  
printf(c=%x\n, c); /\*输出字符的ASCII码值c=41\*/  
上面结果中的地址值在不同计算机上可能不同。

指令 用途  
# 空指令，无任何效果  
#include 包含一个源代码文件  
#define 定义宏  
#undef 取消已定义的宏  
#if 如果给定条件为真，则编译下面代码  
#ifdef 如果宏已经定义，则编译下面代码  
#ifndef 如果宏没有定义，则编译下面代码  
#elif 如果前面的#if给定条件不为真，当前条件为真，则编译下面代码  
#endif 结束一个#if……#else条件编译块  
#error 停止编译并显示错误信息

一、文件包含  
#include预处理指令的作用是在指令处展开被包含的文件。包含可以是多重的，也就是说一个被包含的文件中还可以包含其他文件。标准C编译器至少支持八重嵌套包含。  
预处理过程不检查在转换单元中是否已经包含了某个文件并阻止对它的多次包含。这样就可以在多次包含同一个头文件时，通过给定编译时的条件来达到不同的效果。例如：  
  
#define AAA  
#include t.c  
#undef AAA  
#include t.c  
  
为了避免那些只能包含一次的头文件被多次包含，可以在头文件中用编译时条件来进行控制。例如：  
/\*my.h\*/  
#ifndef MY\_H  
#define MY\_H  
……  
#endif  
  
在程序中包含头文件有两种格式：  
#include < my.h >  
#include “my.h” //常用这种吧  
第一种方法是用尖括号把头文件括起来。这种格式告诉预处理程序在编译器自带的或外部库的头文件中搜索被包含的头文件。第二种方法是用双引号把头文件括起来。这种格式告诉预处理程序在当前被编译的应用程序的源代码文件中搜索被包含的头文件，如果找不到，再搜索编译器自带的头文件。  
采用两种不同包含格式的理由在于，编译器是安装在公共子目录下的，而被编译的应用程序是在它们自己的私有子目录下的。一个应用程序既包含编译器提供的公共头文件，也包含自定义的私有头文件。采用两种不同的包含格式使得编译器能够在很多头文件中区别出一组公共的头文件。  
  
二、宏  
宏定义了一个代表特定内容的标识符。预处理过程会把源代码中出现的宏标识符替换成宏定义时的值。宏最常见的用法是定义代表某个值的全局符号。宏的第二种用法是定义带参数的宏，这样的宏可以象函数一样被调用，但它是在调用语句处展开宏，并用调用时的实际参数来代替定义中的形式参数。  
1.#define指令  
#define预处理指令是用来定义宏的。该指令最简单的格式是：首先神明一个标识符，然后给出这个标识符代表的代码。在后面的源代码中，就用这些代码来替代该标识符。这种宏把程序中要用到的一些全局值提取出来，赋给一些记忆标识符。  
#define MAX\_NUM 10  
int array[MAX\_NUM];  
for(i=0;i  
  
在这个例子中，对于阅读该程序的人来说，符号MAX\_NUM就有特定的含义，它代表的值给出了数组所能容纳的最大元素数目。程序中可以多次使用这个值。作为一种约定，习惯上总是全部用大写字母来定义宏，这样易于把程序红的宏标识符和一般变量标识符区别开来。如果想要改变数组的大小，只需要更改宏定义并重新编译程序即可。  
宏表示的值可以是一个常量表达式，其中允许包括前面已经定义的宏标识符。例如：  
#define ONE 1  
#define TWO 2  
#define THREE (ONE+TWO)  
注意上面的宏定义使用了括号。尽管它们并不是必须的。但出于谨慎考虑，还是应该加上括号的。例如：  
six=THREE\*TWO;  
预处理过程把上面的一行代码转换成：  
six=(ONE+TWO)\*TWO;  
如果没有那个括号，就转换成six=ONE+TWO\*TWO;了。  
宏还可以代表一个字符串常量，例如：  
#define VERSION Version 1.0 Copyright(c) 2003  
2.带参数的#define指令  
带参数的宏和函数调用看起来有些相似。看一个例子：  
#define Cube(x) (x)\*(x)\*(x)  
可以时任何数字表达式甚至函数调用来代替参数x。这里再次提醒大家注意括号的使用。宏展开后完全包含在一对括号中，而且参数也包含在括号中，这样就保证了宏和参数的完整性。看一个用法：  
int num=8+2;  
volume=Cube(num);  
展开后为(8+2)\*(8+2)\*(8+2);  
如果没有那些括号就变为8+2\*8+2\*8+2了。  
下面的用法是不安全的：  
volume=Cube(num++);  
如果Cube是一个函数，上面的写法是可以理解的。但是，因为Cube是一个宏，所以会产生副作用。这里的擦书不是简单的表达式，它们将产生意想不到的结果。它们展开后是这样的：  
volume=(num++)\*(num++)\*(num++);  
很显然，结果是10\*11\*12,而不是10\*10\*10;  
那么怎样安全的使用Cube宏呢？必须把可能产生副作用的操作移到宏调用的外面进行：  
int num=8+2;  
volume=Cube(num);  
num++;  
3.#运算符  
出现在宏定义中的#运算符把跟在其后的参数转换成一个字符串。有时把这种用法的#称为字符串化运算符。例如：  
  
#define PASTE(n) adhfkj#n  
  
main()  
{  
printf(%s\n,PASTE(15));  
}  
宏定义中的#运算符告诉预处理程序，把源代码中任何传递给该宏的参数转换成一个字符串。所以输出应该是adhfkj15。  
4.##运算符  
##运算符用于把参数连接到一起。预处理程序把出现在##两侧的参数合并成一个符号。看下面的例子：  
  
#define NUM(a,b,c) a##b##c  
#define STR(a,b,c) a##b##c  
  
main()  
{  
printf(%d\n,NUM(1,2,3));  
printf(%s\n,STR(aa,bb,cc));  
}  
  
最后程序的输出为:  
123  
aabbcc  
千万别担心，除非需要或者宏的用法恰好和手头的工作相关，否则很少有程序员会知道##运算符。绝大多数程序员从来没用过它。  
  
**三、条件编译指令**  
条件编译指令将决定那些代码被编译，而哪些是不被编译的。可以根据表达式的值或者某个特定的宏是否被定义来确定编译条件。  
1.#if指令  
#if指令检测跟在制造另关键字后的常量表达式。如果表达式为真，则编译后面的代码，知道出现#else、#elif或#endif为止；否则就不编译。  
2.#endif指令  
#endif用于终止#if预处理指令。  
  
#define DEBUG 0  
main()  
{  
#if DEBUG  
printf(Debugging\n);  
#endif  
printf(Running\n);  
}  
  
由于程序定义DEBUG宏代表0，所以#if条件为假，不编译后面的代码直到#endif，所以程序直接输出Running。  
如果去掉#define语句，效果是一样的。  
3.#ifdef和#ifndef  
#define DEBUG  
  
main()  
{  
#ifdef DEBUG  
printf(yes\n);  
#endif  
#ifndef DEBUG  
printf(no\n);  
#endif  
}  
#if defined等价于#ifdef; #if !defined等价于#ifndef  
4.#else指令  
#else指令用于某个#if指令之后，当前面的#if指令的条件不为真时，就编译#else后面的代码。#endif指令将中指上面的条件块。  
  
#define DEBUG  
  
main()  
{  
#ifdef DEBUG  
printf(Debugging\n);  
#else  
printf(Not debugging\n);  
#endif  
printf(Running\n);  
}  
  
5.#elif指令  
#elif预处理指令综合了#else和#if指令的作用。  
  
#define TWO  
  
main()  
{  
#ifdef ONE  
printf(1\n);  
#elif defined TWO  
printf(2\n);  
#else  
printf(3\n);  
#endif  
}  
程序很好理解，最后输出结果是2。  
  
6.其他一些标准指令  
#error指令将使编译器显示一条错误信息，然后停止编译。  
#line指令可以改变编译器用来指出警告和错误信息的文件号和行号。  
#pragma指令没有正式的定义。编译器可以自定义其用途。典型的用法是禁止或允许某些烦人的警告信息。