1. #include <stdio.h>
3. **void** main()
4. {
5. **int** a=12;
6. printf("a的值为:%d\n",a);
7. \_\_asm {mov dword ptr [ebp-4], 0h}
8. **int** b = a;
9. printf("b的值为:%d\n",b);
10. }

使用Debug模式的结果为：

**[cpp]** [view plain](https://blog.csdn.net/bigloomy/article/details/6645810) [copy](https://blog.csdn.net/bigloomy/article/details/6645810)

1. a的值为:12
2. b的值为:0
3. Press any key to **continue**

使用Release模式的结果为：

**[cpp]** [view plain](https://blog.csdn.net/bigloomy/article/details/6645810) [copy](https://blog.csdn.net/bigloomy/article/details/6645810)

1. a的值为:12
2. b的值为:12
3. Press any key to **continue**

由于在Debug模式下我们并没有对代码进行优化，所以对于在代码中每次使用a值得时候都是从它的内存地址直接读取的，所以在我们使用了\_\_asm {mov dword ptr [ebp-4], 0h}语句改变了a的值之后，接下来使用a值的时候从内存中直接读取，所以得到的是更新后的a值；

但是当我们在Release模式下运行的时候，发现b的值为a之前的值，而不是我们更新后的a值，这是由于编译器在优化的过程中做了优化处理。编译器发现在对a赋值之后没有再次改变a的值，所以编译器把a的值备份在了一个寄存器中，在之后的操作中我们再次使用a值的时候就直接操作这个寄存器，而不去读取a的内存地址，因为读取寄存器的速度要快于直接读取内存的速度。这就使得了读到的a值为之前的12。而不是更新后的0。

第二段代码中我们使用了一个volatile修饰符，这种情况下不管在什么模式下都得到的是更新后的a的值，因为volatile修饰符的作用就是告诉编译器不要对它所修饰的变量进行任何的优化，每次取值都要直接从内存地址得到。从这儿我们可以看出，对于我们代码中的那些易变量，我们最好使用volatile修饰，以此来得到每次对其进行更新后的值。

为了加深下大家的印象我们再来看看下面一段代码。

1. #include <stdio.h>
2. #include <sys/time.h>
4. **int** main(**int** argc, **char** \* argv[])
5. {
6. **struct** timeval start,end;
7. gettimeofday( &start, NULL );  /\*测试起始时间\*/
8. **double** timeuse;
9. **int** j;
10. **for**(j=0;j<10000000;j++)
11. ;
12. gettimeofday( &end, NULL );   /\*测试终止时间\*/
13. timeuse = 1000000 \* ( end.tv\_sec - start.tv\_sec ) + end.tv\_usec -start.tv\_usec;
14. timeuse /= 1000000;
15. printf("运行时间为：%f\n",timeuse);
17. **return** 0;
19. }

与之前我们测试时间的代码一样，我们只是增大了for()循环的次数。

先来看看我们不使用优化的结果：

**[cpp]** [view plain](https://blog.csdn.net/bigloomy/article/details/6645810) [copy](https://blog.csdn.net/bigloomy/article/details/6645810)

1. root@ubuntu:/home# gcc time.c -o p
2. root@ubuntu:/home# ./p
3. 运行时间为：0.028260

使用了优化的运行结果：

**[cpp]** [view plain](https://blog.csdn.net/bigloomy/article/details/6645810) [copy](https://blog.csdn.net/bigloomy/article/details/6645810)

1. root@ubuntu:/home# gcc -o p time.c -O2
2. root@ubuntu:/home# ./p
3. 运行时间为：0.000001

从结果显然可以看出差距如此之大，但是如果我们在上面的代码中修改一下int j为int  volatile j之后再来看看如下代码：

**[cpp]** [view plain](https://blog.csdn.net/bigloomy/article/details/6645810) [copy](https://blog.csdn.net/bigloomy/article/details/6645810)

1. #include <stdio.h>
2. #include <sys/time.h>
4. **int** main(**int** argc, **char** \* argv[])
5. {
6. **struct** timeval start,end;
7. gettimeofday( &start, NULL );  /\*测试起始时间\*/
8. **double** timeuse;
9. **int** **volatile** j;
10. **for**(j=0;j<10000000;j++)
11. ;
12. gettimeofday( &end, NULL );   /\*测试终止时间\*/
13. timeuse = 1000000 \* ( end.tv\_sec - start.tv\_sec ) + end.tv\_usec -start.tv\_usec;
14. timeuse /= 1000000;
15. printf("运行时间为：%f\n",timeuse);
17. **return** 0;
19. }

先来看看我们不使用优化的运行结果为：

**[cpp]** [view plain](https://blog.csdn.net/bigloomy/article/details/6645810) [copy](https://blog.csdn.net/bigloomy/article/details/6645810)

1. root@ubuntu:/home# gcc time.c -o p
2. root@ubuntu:/home# ./p
3. 运行时间为：0.027647

使用了优化的运行结果为：

**[cpp]** [view plain](https://blog.csdn.net/bigloomy/article/details/6645810) [copy](https://blog.csdn.net/bigloomy/article/details/6645810)

1. root@ubuntu:/home# gcc -o p time.c -O2
2. root@ubuntu:/home# ./p
3. 运行时间为：0.027390

我们发现此时此刻不管是否使用优化语句运行，时间几乎没有变化，只是有微小的差异，这微小的差异是由于计算机本身所导致的。所以我们通过对于上面一个没有使用volatile和下面一个使用了volatile的对比结果可知，使用了volatile的变量在使用优化语句是for()循环并没有得到优化，因为for()循环执行的是一个空操作，那么通常情况下使用了优化语句使得这个for()循环被优化掉，根本就不执行。就好比编译器在编译的过程中将i的值设置为大于或者等于10000000的一个数，使得for()循环语句不会执行。但是由于我们使用了volatile，使得编译器就不会自作主张的去动我们的i值，所以循环体得到了执行。举这个例子的原因是要让读者牢记，如果我们定义了volatile变量，那么它就不会被编译器所优化。