别。例如，以命令行选项“-Og”调用 GCC 是让 GCC 使用一组基本的优化。以选项“-O1”或更高(如“-O2”或“-O3”)调用 GCC 会让它使用更大量的优化。这样做可以进一步提高程序的性能，但是也可能增加程序的规模，也可能使标准的调试工具更难对程序进行调试。我们的表述，虽然对于大多数使用 GCC 的软件项目来说，优化级别-O2 已经成为了被接受的标准，但是还是主要考虑以优化级别-O1 编译出的代码。我们特意限制了优化级别，以展示写 C 语言函数的不同方法如何影响编译器产生代码的效率。我们会发现可以写出的 C 代码，即使用-O1 选项编译得到的性能，也比用可能的最高的优化等级编译一个更原始的版本得到的性能好。

编译器必须很小心地对程序只使用安全的优化，也就是说对于程序可能遇到的所有可能的情况，在 C 语言标准提供的保证之下，优化后得到的程序和未优化的版本有一样的行为。限制编译器只进行安全的优化，消除了造成不希望的运行时行为的一些可能的原因，但是这也意味着程序员必须花费更大的力气写出编译器能够将之转换成有效机器代码的程序。为了理解决定一种程序转换是否安全的难度，让我们来看看下面这两个过程：

```
1   void twiddle1(long *xp, long *yp)
2   {
3       *xp += *yp;
4       *xp += *yp;
5   }
6
7   void twiddle2(long *xp, long *yp)
8   {
9       *xp += 2* *yp;
10  }
```

乍一看，这两个过程似乎有相同的行为。它们都是将存储在由指针 yp 指示的位置处的值两次加到指针 xp 指示的位置处的值。另一方面，函数 twiddle2 效率更高一些。它只要求 3 次内存引用(读*xp，读*yp，写*xp)，而 twiddle1 需要 6 次(2 次读*xp，2 次读*yp，2 次写*xp)。因此，如果要编译器编译过程 twiddle1，我们会认为基于 twiddle2 执行的计算能产生更有效的代码。

不过，考虑 xp 等于 yp 的情况。此时，函数 twiddle1 会执行下面的计算：

```
3       *xp += *xp;  /* Double value at xp */
4       *xp += *xp;  /* Double value at xp */
```

结果是 xp 的值增加 4 倍。另一方面，函数 twiddle2 会执行下面的计算：

```
9       *xp += 2* *xp;  /* Triple value at xp */
```

结果是 xp 的值增加 3 倍。编译器不知道 twiddle1 会如何被调用，因此它必须假设参数 xp 和 yp 可能会相等。因此，它不能产生 twiddle2 风格的代码作为 twiddle1 的优化版本。

这种两个指针可能指向同一个内存位置的情况称为*内存别名使用*(memory aliasing)。在只执行安全的优化中，编译器必须假设不同的指针可能会指向内存中同一个位置。再看一个例子，对于一个使用指针变量 p 和 q 的程序，考虑下面的代码序列：

```
x = 1000; y = 3000;
*q = y;   /* 3000 */
*p = x;   /* 1000 */
t1 = *q;  /* 1000 or 3000 */
```