


t1 的计算值依赖于指针 p 和 q 是否指向内存中同一个位置——如果不是, t1 就等于 3000, 但如果是, t1 就等于 1000。这造成了一个主要的妨碍优化的因素, 这也是可能严重限制编译器产生优化代码机会的程序的一个方面。如果编译器不能确定两个指针是否指向同一个位置, 就必须假设什么情况都有可能, 这就限制了可能的优化策略。

 **练习题 5.1** 下面的问题说明了内存别名使用可能会导致意想不到的程序行为的方式。考虑下面这个交换两个值的过程:

```
1  /* Swap value x at xp with value y at yp */
2  void swap(long *xp, long *yp)
3  {
4      *xp = *xp + *yp; /* x+y */
5      *yp = *xp - *yp; /* x+y-y = x */
6      *xp = *xp - *yp; /* x+y-x = y */
7  }
```

如果调用这个过程时 xp 等于 yp, 会有什么样的效果?

第二个妨碍优化的因素是函数调用。作为一个示例, 考虑下面这两个过程:

```
1  long f();
2
3  long func1() {
4      return f() + f() + f() + f();
5  }
6
7  long func2() {
8      return 4*f();
9  }
```

最初看上去两个过程计算的都是相同的结果, 但是 func2 只调用 f 一次, 而 func1 调用 f 四次。以 func1 作为源代码时, 会很想产生 func2 风格的代码。

不过, 考虑下面 f 的代码:

```
1  long counter = 0;
2
3  long f() {
4      return counter++;
5  }
```

这个函数有个副作用——它修改了全局程序状态的一部分。改变调用它的次数会改变程序的行为。特别地, 假设开始时全局变量 counter 都设置为 0, 对 func1 的调用会返回 $0+1+2+3=6$, 而对 func2 的调用会返回 $4 \cdot 0=0$ 。

大多数编译器不会试图判断一个函数是否没有副作用, 如果没有, 就可能被优化成像 func2 中的样子。相反, 编译器会假设最糟的情况, 并保持所有的函数调用不变。

旁注 用内联函数替换优化函数调用

包含函数调用的代码可以用一个称为内联函数替换 (inline substitution, 或者简称“内联 (inlining)”) 的过程进行优化, 此时, 将函数调用替换为函数体。例如, 我们可以通过替换掉对函数 f 的四次调用, 展开 func1 的代码:

```
1  /* Result of inlining f in func1 */
2  long func1in() {
3      long t = counter++; /* +0 */
```