这段无危险的代码变成了一个主要的性能瓶颈。相比较而言，lower2 的性能对于任意长度的字符串来说都是足够的。大型编程项目中出现这样问题的故事比比皆是。一个有经验的程序员工作的一部分就是避免引入这样的渐近低效率。

练习题 5.3　考虑下面的函数：

```
long min(long x, long y) { return x < y ? x : y; }
long max(long x, long y) { return x < y ? y : x; }
void incr(long *xp, long v) { *xp += v; }
long square(long x) { return x*x; }
```

下面三个代码片断调用这些函数：

A.
```
for (i = min(x, y); i < max(x, y); incr(&i, 1))
    t += square(i);
```

B.
```
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);
```

C.
```
long low = min(x, y);
long high = max(x, y);

for (i = low; i < high; incr(&i, 1))
    t += square(i);
```

假设 x 等于 10，而 y 等于 100。填写下表，指出在代码片断 A～C 中 4 个函数每个被调用的次数：

代码	min	max	incr	square
A.				
B.				
C.				

5.5　减少过程调用

像我们看到过的那样，过程调用会带来开销，而且妨碍大多数形式的程序优化。从 combine2 的代码(见图 5-6)中我们可以看出，每次循环迭代都会调用 get_vec_element 来获取下一个向量元素。对每个向量引用，这个函数要把向量索引 i 与循环边界做比较，很明显会造成低效率。在处理任意的数组访问时，边界检查可能是个很有用的特性，但是对 combine2 代码的简单分析表明所有的引用都是合法的。

作为替代，假设为我们的抽象数据类型增加一个函数 get_vec_start。这个函数返回数组的起始地址，如图 5-9 所示。然后就能写出此图中 combine3 所示的过程，其内循环里没有函数调用。它没有用函数调用来获取每个向量元素，而是直接访问数组。一个纯粹主义者可能会说这种变换严重损害了程序的模块性。原则上来说，向量抽象数据类型的使用者甚至不应该需要知道向量的内容是作为数组来存储的，而不是作为诸如链表之类的某种其他数据结构来存储的。比较实际的程序员会争论说这种变换是获得高性能结果的必要步骤。

函数	方法	整数		浮点数	
		+	*	+	*
combine2	移动 vec_length	7.02	9.03	9.02	11.03
combine3	直接数据访问	7.17	9.02	9.02	11.03