后，无论是什么样的高速缓存结构，对 v 的引用都会得到下面的命中和不命中模式：

v[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$
访问顺序，命中[h]或不命中[m]	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]

在这个例子中，对 v[0]的引用会不命中，而相应的包含 v[0]～v[3]的块会被从内存加载到高速缓存中。因此，接下来三个引用都会命中。对 v[4]的引用会导致不命中，而一个新的块被加载到高速缓存中，接下来的三个引用都命中，依此类推。总的来说，四个引用中，三个会命中，在这种冷缓存的情况下，这是我们所能做到的最好的情况了。

总之，简单的 sumvec 示例说明了两个关于编写高速缓存友好的代码的重要问题：

- 对局部变量的反复引用是好的，因为编译器能够将它们缓存在寄存器文件中(时间局部性)。
- 步长为 1 的引用模式是好的，因为存储器层次结构中所有层次上的缓存都是将数据存储为连续的块(空间局部性)。

在对多维数组进行操作的程序中，空间局部性尤其重要。例如，考虑 6.2 节中的 sumarrayrows 函数，它按照行优先顺序对一个二维数组的元素求和：

```
1   int sumarrayrows(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
6           for (j = 0; j < N; j++)
7               sum += a[i][j];
8       return sum;
9   }
```

由于 C 语言以行优先顺序存储数组，所以这个函数中的内循环有与 sumvec 一样好的步长为 1 的访问模式。例如，假设我们对这个高速缓存做与对 sumvec 一样的假设。那么对数组 a 的引用会得到下面的命中和不命中模式：

a[i][j]	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
$i=0$	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
$i=1$	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
$i=2$	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
$i=3$	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

但是如果我们做一个看似无伤大雅的改变——交换循环的次序，看看会发生什么：

```
1   int sumarraycols(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (j = 0; j < N; j++)
6           for (i = 0; i < M; i++)
7               sum += a[i][j];
8       return sum;
9   }
```

在这种情况中，我们是一列一列而不是一行一行地扫描数组的。如果我们够幸运，整个数组都在高速缓存中，那么我们也会有相同的不命中率 1/4。不过，如果数组比高速缓存要