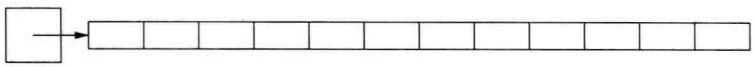
Matrix<2,double> 的内存布局则如下所示：



为构造 vector<vector<double>>，我们需要调用四次构造函数并进行四次自由存储空间分配操作。为访问元素，我们需要做两次间接寻址。

为构造 Matrix<2,double>，我们只需调用一次构造函数、进行一次自由存储空间分配。为访问元素，也只需做一次间接寻址。

当到达某行中的一个元素后，为访问其前驱元素我们不必再进行一次间接访问，因此 vector<vector<double>> 的元素访问代价并不总是 Matrix<2,double> 的两倍。但是对要求高性能的算法而言，vector<vector<double>> 的链接结构所产生的分配、释放和访问代价仍然是一个问题。

vector<vector<double>> 解决方案隐含了一个可能性：每行的大小可以不同。有些时候这的确是一个优点，但大多数情况下这只是为错误提供了机会、为测试增加了负担。

当我们需要更高维矩阵时，上述问题和额外开销会变得更严重：你可以比较 vector<vector<vector<double>>> 相对于 Matrix<3,double> 增加的间接寻址和分配操作次数。

总之，我注意到数据结构紧凑存储的重要性经常被低估或妥协。紧凑存储的优点既是逻辑上的也是关乎性能的。综合考虑它被低估及指针和 new 被过度使用的趋势，我们现在面临一个广泛存在的问题。例如，考虑在实现一个二维结构时将每行实现为自由存储空间上的独立对象 vector<vector<double>*>，这会带来开发复杂性、运行时代价、内存开销以及错误可能等诸多问题。

31.4.1.3　vector 和数组

vector 是一种资源句柄，这是允许改变大小和实现高效移动语义的原因。但是，这一特点偶尔也会变为缺点——尤其是与不依赖于元素和句柄分离存储的数据结构（如内置数组和 array）相比。将元素序列保存在栈中或另一个对象中可能会带来性能上的优势，就像它也可能是劣势一样。

vector 能很好地处理初始化后的对象。这令我们使用 vector 变得很简单并能依赖元素的恰当析构操作。但是，这一特点偶尔也会变成缺点——尤其是与允许未初始化元素的数据结构（如内置数组和 array）相比。

例如，在向数组元素存入数据之前，我们不必初始化它们：

```
void read()
{
    array<int,MAX]> a;
    for (auto& x : a)
        cin.get(&x);
}
```

对 vector，我们可以用 emplace_back() 达到类似效果（但不必指定一个 MAX）。

31.4.1.4　vector 和 string

vector<char> 是可改变大小、连续存储的 char 序列，string 也是如此。那么我们应该如何在两者间进行选择呢？

vector 是一种保存值的通用机制，并不对保存的值之间的关系做任何假设。对一个 vector<char> 而言，字符串 Hello, World! 只不过是一个 13 个 char 类型的元素的序列而已。